

### 版权相关注意事项：

- 1、书籍版权归著者和出版社所有
- 2、本PDF来自于各个广泛的信息平台，经过整理而成
- 3、本PDF仅限用于非商业用途或者个人交流研究学习使用
- 4、本PDF获得者不得在互联网上以任何目的进行传播
- 5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
- 6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
- 7、请于下载PDF后24小时内研究使用并删掉本PDF



Web开发经典丛书

使用React、JSX、Redux和GraphQL开发Web Apps

# 快速上手 React 编程

John Sonmez  
作序推荐本书



[美] 阿扎·马尔丹 (Azat Mardan) 著

郭美青 郭 松 唐金州 译

Web 开发经典丛书

# 快速上手 React 编程

[美] 阿扎·马尔丹 (Azat Mardan) 著

郭美青 郭 松 唐金州 译

清华大学出版社

北 京





Azat Mardan

React Quickly

EISBN: 978-1-61729-334-4

Original English language edition published by Manning Publications, 178 South Hill Drive, Westampton, NJ 08060 USA. Copyright © 2017 by Manning Publications. Simplified Chinese-language edition copyright © 2018 by Tsinghua University Press. All rights reserved.

本书中文简体字版由 Manning 出版公司授权清华大学出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

北京市版权局著作权合同登记号 图字：01-2017-7950

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目(CIP)数据

快速上手 React 编程 / (美)阿扎·马尔丹(Azat Mardan) 著；郭美青，郭松，唐金州 译. —北京：清华大学出版社，2018

(Web 开发经典丛书)

书名原文：React Quickly

ISBN 978-7-302-50247-0

I. ①O… ②快… II. ①阿… ②郭… ③郭… ④唐… III. ①移动终端—应用程序—程序设计  
IV. ①TN929.53

中国版本图书馆 CIP 数据核字(2018)第 103232 号

责任编辑：王 军 李维杰

装帧设计：思创景点

责任校对：孔祥峰

责任印制：刘海龙

出版发行：清华大学出版社

网 址：<http://www.tup.com.cn>, <http://www.wqbook.com>

地 址：北京清华大学学研大厦 A 座 邮 编：100084

社 总 机：010-62770175 邮 购：010-62786544

投稿与读者服务：010-62776969, [c-service@tup.tsinghua.edu.cn](mailto:c-service@tup.tsinghua.edu.cn)

质 量 反 馈：010-62772015, [zhiliang@tup.tsinghua.edu.cn](mailto:zhiliang@tup.tsinghua.edu.cn)

印 装 者：清华大学印刷厂

经 销：全国新华书店

开 本：185mm×260mm 印 张：28.75 字 数：700 千字

版 次：2018 年 6 月第 1 版 印 次：2018 年 6 月第 1 次印刷

印 数：1~3500

定 价：88.00 元

产品编号：075962-01



## 本书赞誉

“对于任何想要获得 React 入门指导，并了解其周边生态系统(包括工具、概念和库)的人来说，本书提供了一种一站式服务。跟随 Azat 的演练，完成给定的项目，你将很快理解 React、Redux、GraphQL、Webpack 和 Jest，以及如何让它们运行起来。”

——Peter Cooper, *JavaScript Weekly* 杂志编辑

“本书通过 React 向读者传授在构建现代 Web 应用的过程中最具价值和值得关注的概念，包括 GraphQL、Webpack 和服务端渲染。阅读之后，你应该有足够的信心能够使用 React 创建一个生产级的 Web 应用。”

——Stan Bershadskiy, *React Native Cookbook* 一书的作者

“Azat 是编程领域最具权威的声音之一。本书深入研究了 React 的基础和架构，并远远超越了基础。对于任何开发者来说，它都是必读的！”

——Erik Hanchett, *Ember.js Cookbook* 一书的作者

“这本书很简单。它使用非常基础的语言让你逐步了解每一个概念。”

——Israel Morales, SavvyCard 前端开发者和 Web 设计师

“简单的语言配以简单的逻辑示例，让你起步并快速运行程序，本书涵盖了任何 React 新手在开始编写应用时都会遇到的主要问题。作者的幽默感会让你一直忙到最后。感谢 Azat 花时间与我们分享他的 React 之旅。”

——Suhas Deshpande, Capital One 软件工程师

“本书是加速学习 React 的好资源，非常透彻和中肯。我将用它作为开发下一个应用的参考。”

——Nathan Bailey, SpringboardAuto.com 全栈开发者

“Azat 非常擅长他所做的事情——教人如何写代码。本书包含基础知识和实例，能让你快速开始使用 React。”

——Shu Liu, IT 顾问

“自 2013 年被 Facebook 开源以来，React 已经迅速成为一个被广泛采用的 JS 库，并且





是 GitHub 上最著名的项目之一。在他的新书《快速上手 React 编程》中，Azat Mardan 以他典型的简明风格，把你需要了解的关于 React 生态系统的一切都编排好了，以便快速构建高性能的 SPA 应用。仅仅关于 React 状态和同构 JavaScript 的章，就值得你购买和阅读。”

—Prakash Sarma, New Star Online

“本书将通过传达清晰的基础知识来让你慢慢接纳 React，它将让你构建应用并彻底接受使用 React 带来的好处。”

—Allan Von Schenkel, FoundHuman 技术与战略副总裁

“本书以易于理解的方式涵盖了 React 的所有重要方面。和 Azat 的所有作品一样，本书清晰而简洁，并且包含快速提高生产力所需的内容。如果你有兴趣将 React 加入自己的技能集，我建议从这里开始。”

—Bruno Watt, hypermedia.tech 咨询架构师

“这是一本关于使用 React 进行全栈 Web 开发的非常全面的书，不仅涵盖 React 本身，还包含周边的生态系统。我总是对 React 服务端渲染感到困惑，Azat 的这本书终于帮我理解了它。如果你是 React 新手并想要真正掌握它，看这本书就足够了。”

—Richard Kho, Capital One 软件工程师



# 译者序

在本书翻译之初，React 社区刚刚经历了两个重量级的事件。一件是前后持续一年有余的 Facebook 附加专利条款事件，后因“React 能不能继续商用”的问题在国内被炒得沸沸扬扬，但随着事件持续发酵，最终迎来了让开发者皆大欢喜的局面；另一件是 React 团队发布了最新的基于 Fiber 架构的 v16.0.0 正式版，带来了众多优秀特性和性能提升。一方面我们看到了 React 无与伦比的业界影响力，甚至连 Facebook 都不惜以得罪开源社区的方式，希望从 React 身上获得商业利益保护的能力；另一方面我们又看到一支孜孜不倦、勇于探索、不断进取的行业先锋团队，当大部分人还在纠结该使用 React、Vue 还是 Angular 2 的时候，React 团队再一次用 Fiber 架构突破浏览器自身的局限，为 React 的发展带来更大的想象空间。React 无疑是当下最热门、最具前景的前端技术之一。

当下，我们正处在“JavaScript 文艺复兴时代”。各种标准、思想、框架、类库百花齐放，各种工具、概念、术语层出不穷。条条大路通罗马，要实现一个前端应用，解决方案除了当年那个永恒的 jQuery 以外，可谓数不胜数，涉及的专有名词更是目不暇接。这是件令人兴奋的事情，因为这意味着前端生态在不断壮大，JavaScript 的能力也在日益增强。但对于前端从业者而言，却是不小的挑战。

依稀记得刚接触 React 时，面对扑面而来的新鲜术语的那种困惑，就好像看见散落一地的珍珠，却不知该如何捡起串成美丽的项链一样。Azat 无疑非常擅长这项工作，他把自己在一线教授 React 的课程、经验、资源以及学员的反馈重新梳理、优化、整合，按照由浅入深、由表及里的编排方式，不仅关注核心，也注重生态，同时配上丰富的实战案例和架构思想解读，为我们带来了这本学习 React 不容错过的《快速上手 React 编程》。相信这本书一定会为读者带来醍醐灌顶般的阅读体验，因为该书不仅仅教你如何使用 React，还告诉你 React 技术体系中每一项设计的原因和思考。希望读者能从中受益，享受学习和使用 React 带来的快乐，这也是译者莫大的荣幸。

本书由郭美青、郭松、唐金州合译完成。特别感谢梁宵的引荐，让我们得以共同合作参与本书的翻译，在整个翻译过程中，他高效的组织、协调能力和审校的专业让我们受益匪浅。感谢我们的家人，你们的支持和鼓励让我们感到无比的温暖。感谢清华大学出版社李阳老师的悉心指导以及在翻译过程中给予的极大理解和帮助。

我们在本书翻译过程中力求做到行文流畅、风格一致，希望能给读者带来愉悦的阅读体验，并尽力修正一些原书中的小错误，但鉴于自身水平有限，疏漏在所难免，敬请广大读者批评指正。

最后，希望本书能帮助业界同仁系统地掌握 React 并使用 React 打造出高性能、更易于维护的 Web 产品。

译者





# 序 言

认真地讲，我一直希望 JavaScript 可以凋零、磨灭。

这并不是说我完全不喜欢 JavaScript——多年以来，它已经改进了不少；而是因为我对复杂的极度厌恶——以至于我将我的博客和公司起名为 Simple Programmer。我的口号一直是：“让复杂变得简单。”

让复杂变得简单并不容易。它需要一套特殊技能。你必须能够理解复杂，并把它理解得很好，这样你才能够从中提炼出核心——因为任何事情在核心上都是简单的。这正是 Azat 在本书中所做的。

现在，我承认 Azat 提供了一点帮助。你能看出，我个人喜欢 React 的原因之一就是它很简单。它被设计得很简单，用于处理越来越复杂的 JavaScript 框架，并通过回归基础(即原始的 JavaScript)来降低复杂性(至少在大多数情况下是这样的。React 的确有一种可以编译成 JavaScript 的 JSX 语言，但是我会让 Azat 介绍给你)。

重点是，尽管我喜欢 Angular、Backbone，以及一些其他的 JavaScript 框架，因为它们能够帮助 Web 开发者更容易地创建异步 Web 应用和单页面应用，但这同时也增加了大量的复杂性。使用模板并理解这些框架的语法和微妙之处能提高生产率，但它们把复杂性从后端转移到了前端。React 一开始就摒弃了模板，而是提供了一种使用 JavaScript 将基于组件的架构应用于 UI 的方式。我喜欢这样。它很简单。但即使最简单的事情也很难解释——或者更糟糕的是，被缺乏这种能力的老师变复杂了。

这正是 Azat 的强项。他知道如何教学，如何简化。他通过解释 React 开始本书，对比了你可能已经知道的 Angular。即使你不了解 Angular，他对 React 的解释也能快速帮助你理解其基本原理和用途。然后，Azat 很快示范了如何创建一个基本的 React 应用，这样你就能了解并亲自实践了。在那之后，他会通过任何人都很容易掌握的真实示例，带你去了解你需要知道的 20%，以便完成你将在 React 中做的剩余 80%。最后——也是我最喜欢的部分——里面包含了丰富的例子和项目。学习的最佳方式绝对是实践，而 Azat 会使用 React 带你去创建 6 个非凡的项目。

为了保持我简单的风格，让我最后说一句：《快速上手 React 编程》提供了我所知道的学习 React 的最佳方法。

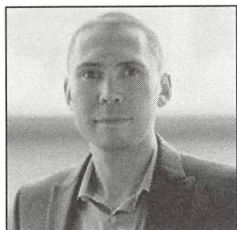
John Sonmez

*Soft Skills* 一书的作者(<http://amzn.to/2hFHxAu>)

Simple Programmer 创始人(<https://simpleprogrammer.com>)



# 作者简介



我已经出版了超过 14 本书和 17 门在线课程(<https://node.university>), 它们中的大多数存储在云端, 涉及 React、JavaScript 和 Node.js。在关注 Node 之前, 我也使用过其他语言编程(Java、C、Perl、PHP、Ruby), 几乎从高中开始编程(十多年前), 并且绝对超过了规定的一万小时。

现在, 我是美国十大银行之一的技术研究员, 它也是一家财富 500 强公司: Capital One Financial Corporation, 位于美丽的旧金山。在那之前, 我曾为小型初创公司、大型企业, 甚至美国联邦政府工作过, 编写桌面、Web 和移动应用, 从事教学、开发协调和项目管理。

我不想占用你太多的时间介绍自己, 你可以在我的博客(<http://webapplog.com/about>)和社交媒体([www.linkedin.com/in/azatm](http://www.linkedin.com/in/azatm))上了解我的更多信息。相反, 我想把关于这本书的经历写下来。

2011 年, 我搬到了阳光明媚的加州, 开始通过企业加速器创业(如果你对此好奇, 它就是 500 家初创公司), 开始使用现代的 JavaScript。我学会了使用 Backbone 为公司创建一些应用, 并对此印象深刻。该框架极大地改进了我前几年构建的单页面应用的代码组织。它有路由和模型。这很棒!

我在 DocuSign(去 Google 搜索一下 e-signatures, 它的市场占有率达到 70%)担任软件工程团队领导时, 又有机会看到了 Backbone 和同构 JavaScript 的惊人能力。我们重构了一个有 7 年历史的 ASP.NET Web 应用, 每一个小版本的发布都需要四周的时间, 而使用时髦的 Backbone-Node-CoffeeScript-Express 应用不仅具有非常好的用户体验, 而且发布版本只需要一到两周的时间。设计团队在可用性方面做得很好。毋庸置疑, 有大量具有不同程度交互的 UI 视图。

最终的应用是同构的, 甚至在此之前还不存在该术语。我们使用服务器上的 Backbone 模型从 API 获取数据并将其缓存。我们在浏览器和服务器上使用相同的 Jade 模板。

这是一个有趣的项目, 它让我更加相信使用一种语言横跨整个技术栈的力量。精通 C# 和前端 JavaScript(大部分是 jQuery)的老款应用开发者, 如果花点时间突击一下(一个发布周期, 通常是一到两周), 就会爱上结构清晰的 CoffeeScript、Backbone 的组织结构以及 Node 的速度(开发和运行速度)。

我在 Web 开发领域十几年的经验向我揭示了好的、坏的以及丑陋的(大部分是丑陋的)前端开发。然而柳暗花明, 因为自从切换到 React, 我就越来越喜欢它了。

如果你想收到更新、新闻和小贴士, 可以在网上通过以下方式联系我, 订阅、加好友、





悄悄关注均可：

- Twitter—[https://twitter.com/azat\\_co](https://twitter.com/azat_co)
- 网站—<http://azat.co>
- LinkedIn—<http://linkedin.com/in/azatm>
- 专业博客—<http://webapplog.com>
- 出版物—<http://webapplog.com/books>

关于面对面研讨会和课程，请访问 <http://NodeProgram.com> 或 <https://Node.University>，或通过 <https://webapplog.com/azat> 给我发消息。



# 致 谢

我要感谢互联网、全世界以及人类的聪明才智，让我们相信心灵感应是可能的。不必开口，就可以通过 Twitter、Facebook 和 Instagram 等社交媒体与全球数百万人分享自己的想法。

我对我的老师心存感激，无论是在中学还是在大学，不经意间，我从书本和耳濡目染的学习中领悟到他们的智慧。

史蒂芬·金说过：“写作是人类做的事情，编辑是上帝做的事情。”因此，我要无数次感谢本书的编辑，更要感谢那些读者，他们在本书中不得不面对不可避免的打字错误和 bug。这是我的第 14 本书，我知道无论怎样都会产生这些问题。

感谢 Manning 出版社的工作人员，他们使本书付梓成为可能：发行人 Marjan Bace 以及所有编辑制作团队，包括 Janet Vail、Kevin Sullivan、Tiffany Taylor、Katie Tennant、Gordan Salinovic、Dan Maharry，还有很多其他的幕后工作者。

我不知如何表达对 Ivan Martinovic 领导的超凡技术审校人团队的感谢：James Anaipakos、Dane Balia、Art Bergquist、Joel Goldfinger、Peter Hampton、Luis Matthew Heck、Ruben J. Leon、Gerald Mack、Kamal Raj 和 Lucas Tettamanti。他们的贡献包括捕获术语中的技术失误、错误、拼写错误，并提供专题建议。每个人都经历了审校过程，论坛主题中的每一条反馈塑造了本书的手稿。

在技术方面，特别感谢 Anto Aravinth，他是本书的技术编辑；还要感谢 German Frigerio，他是本书的技术校对。他们正是我所期望的最好的技术编辑。

非常感谢 John Sonmez 为本书作序，他闻名于 Pluralsight、Manning 出版社以及 SimpleProgrammer.com。感谢 Peter Cooper、Erik Hanchett 和 Stan Bershadskiy 所做的审校工作，你们赋予本书额外的可信度。没有听说过 John、Peter、Erik 或 Stan 的读者应该关注他们在软件工程领域所做的工作。

最后，感谢所有 MEAP 读者的反馈。基于你们的评论，我修订了本书，虽然使本书出版延迟了一年，但却造就了当下关于 React 的一本好书。



# 前言

那是 2008 年，银行纷纷倒闭。我在联邦存款保险公司(Federal Deposit Insurance Corporation, FDIC)工作，主要任务是偿还倒闭、失败和破产的银行储户。我承认，就职业安全感而言，我的工作等同于在雷曼兄弟上班或在泰坦尼克号上卖票。但是，当我所在的部门还没有开始最终的预算削减时，我还有机会开发一款名为 EDIE 的应用。这款应用由于一个简单的原因而变得非常流行：人们急于想知道他们的存款中有多少由美国联邦政府提供保险，而 EDIE 能够估算这一数额。

但是存在一个问题：人们不喜欢把他们的私人账户告诉政府。为了保护他们的隐私，这款应用完全在前端完成：JavaScript、HTML 和 CSS，没有任何后端技术。如此，FDIC 就不会收集任何个人财产信息。

这款应用经历了数十次迭代，留下了一团混乱的意大利面条式代码。开发人员来去匆匆，没有留下任何文档，也没有任何符合逻辑的简单算法。这就像没有地图就要去乘坐纽约市地铁一样。有很多函数可以调用其他函数、奇怪的数据结构以及更多的函数。在现代术语中，这款应用采用纯用户界面(User Interface, UI)，因为没有后端。

我多么希望那时能有 React。React 带来了愉悦。它是一种全新的思考方式，也是一种全新的开发方式。将核心功能放在一处，而不是将其分解为 HTML 和 JS，这种简单是一种解放。它重新点燃了我对前端开发的热情。

React 是开发 UI 组件的一种新方法。它是新一代的表现层库。与模型和路由库一起配合，React 可以在 Web 和移动技术栈中取代 Angular、Backbone 或 Ember。这就是我写这本书的原因。我从来不喜欢 Angular：它太复杂和大一统了。模板引擎是特定领域的，甚至不再是 JavaScript；它是另一种语言。我使用过 Backbone，喜欢它的简单和 DIY 方式。Backbone 是成熟的，它更像是你自己框架的基石，而不是一个全面的、大一统的框架。Backbone 的问题是模型与视图之间不断增加的交互复杂性：多个视图更新多种模型，有的会更新其他视图，有的会触发模型上的事件。

举办 React 在线课程(<http://mng.bz/XgkO>)，以及参加各种会议和活动的个人经验已经显示，开发者渴望一种更好的开发 UI 的方法。大多数商业价值存在于 UI，而后端负责提供商品。在海湾地区，也是我居住和工作的地方，大多数软件工程师的职位空缺是前端或(一个时髦的头衔)通才/全栈开发者。只有少数像谷歌、亚马逊和 Capital One 这样的大公司，对数据科学家和后端工程师的需求依然强劲。

想要确保职业安全感，或找到一份好工作，最好的方法就是成为一名多面手。要做到这一点，最快的方法是使用同构的、可扩展的、对开发者友好的库，就像前端的 React 搭配后端的 Node.js，以防止需要处理服务器端代码。

对于移动开发者来说,HTML5 是两三年以前的一个肮脏词汇。Facebook 放弃了 HTML5 应用,转而支持更高效的原生(native)实现。但是这种不利观点正在迅速改变。通过 React Native,可以渲染移动应用:可以保留 UI 组件,但在不同的环境下调整它们,这是另一种支持学习 React 的观点。

编程可以是创造性的。不要陷入乏味的任务、复杂的事情以及伪关注点分离。砍掉所有不必要的垃圾,通过 React 提供的简化的模块之美、基于组件的 UI 释放你的创造力。为同构的 JavaScript 引入一些 Node,你将达到禅定的境界。

祝你阅读愉快,并让我知道你对这本书的评价,请在 Amazon.com(<http://amzn.to/2gPxxv9Q>)上留言。



# 学习建议

本书旨在解决前端开发者的烦恼，使他们的生活更有意义、更加快乐，并通过介绍 React 帮助他们赚到更多的钱——这些会以一种快速的方式进行。本书是十几个人工作一年半的成果。至少，本书会开拓你的思维，让你了解一些不寻常的概念，如 JSX、单向数据流和声明式编程。

## 路线图

本书分为两部分：“React 核心”（第 1 至第 11 章）和“React 生态”（第 12 至第 20 章）。每一章都会为描述性的文字配以恰当的代码示例和图表。每一章也都有一段可选的介绍视频，帮助你决定是否需要阅读该章。各章是以独立的方式编写的，这意味着即便不按顺序阅读，也不会带来麻烦——尽管建议按顺序阅读。每章的结尾是一个小测验，以强化你对该章内容的记忆，最后还有小结。

每一部分都以一系列更大的项目结尾，这些项目将给你带来更多的 React 经验，通过前几章介绍的概念和知识构建应用，巩固你对知识的理解。这些项目以可选的截屏视频作为补充来强化你的学习，向你展示动态的过程，如创建文件和安装依赖（Web 开发包含很多组成部分！）。这些项目是书中不可或缺的一部分——不要跳过它们。建议你自己键入每一行代码，并避免复制和粘贴。研究表明：打字和写作能提高学习效率。

本书以 5 个附录为结尾，它们提供了补充材料。在开始阅读之前，请连同目录一起把它们看一遍。

本书的网站是 [www.manning.com/books/react-quickly](http://www.manning.com/books/react-quickly) 和 <http://reactquickly.co>。如果需要最新的信息，很可能会在那里找到。

源代码可以在 Manning 出版社的网站 ([www.manning.com/books/react-quickly](http://www.manning.com/books/react-quickly)) 和 GitHub (<https://github.com/azat-co/react-quickly>) 上找到。查看“源代码”部分可获得更多详情。读者也可扫描本书封底的二维码来下载源代码和每章的介绍视频。书中已展示完整的代码清单——这比跳转到 GitHub 或代码编辑器去查看文件方便得多。

## 本书适合的读者(必读！)

本书适合 Web 和移动开发者以及具备两到三年工作经验的软件工程师阅读，他们想开始学习并使用 React，用于 Web 或移动开发。基本上，本书适合那些对开发者工具的快捷



键了然于心的人(比如 Mac 上的 `Cmd+Opt+J` 或 `Cmd+Opt+I`)。本书的目标读者是知晓并熟悉以下概念的人:

- 单页面应用(Single Page Application, SPA)
- RESTful 服务和 API 架构
- JavaScript, 尤其包括闭包、作用域, 以及字符串和数组方法
- HTML、HTML5, 以及它们的元素和属性
- CSS 及其样式, 以及 JavaScript 选择器

有 jQuery、Angular、Ember、Backbone 或其他 MVC 类框架的使用经验是一个加分项, 因为你可以用 React 的方式来对比它们。但这是没有必要的, 在某种程度上甚至可能是有害的, 因为你需要忘掉某些模式。React 并不完全是 MVC。

你将使用命令行工具, 如果害怕它们, 那么这是对抗命令行/终端/命令提示符恐惧的最佳时机。一般来说, CLI 比它们的可视化(GUI)版本更加强大和通用(比如 Git 的命令行版及桌面版, 后者把我搞糊涂了)。

熟悉 Node.js 会让你比从未听说过 Node.js、npm、Browserify、CommonJS、Gulp 或 Express.js 的人学得更快。我写过几本关于 Node.js 的书, 对于那些想要重温它的人, 最流行的就是 *Practical Node.js*(<http://practicalnodebook.com>)。或者, 你也可以去网上寻找免费的 NodeSchool 课程(<http://nodeschool.io>)(免费并不总是意味着更糟)。

## 本书未包含的内容(也请必读!)

本书并不是一本全面的 Web 或移动开发指南。本书假设你已经了解这些知识。如果你想学习基本的编程概念或 JavaScript 基础知识, 有很多关于这些主题的好书。我想到的有: Kyle Simpson 的 *You Don't Know Js*(<https://github.com/getify/You-Dont-Know-JS>)、*Secrets of the JavaScript Ninja, Second Edition*([www.manning.com/books/secrets-of-the-javascript-ninja-second-edition](http://www.manning.com/books/secrets-of-the-javascript-ninja-second-edition)), 以及 Marijn Haverbeke 的 *Eloquent JavaScript*(<http://eloquentjavascript.net>)。所以, 我没必要在本书中重复已有的内容。

## 如何使用本书

首先, 你应该阅读这本书。这不是开玩笑。大多数人把书买回来, 但从不去读。阅读这本书吧, 并逐章去完成那些项目。

每一章都涉及一个或一系列主题, 它们互为基础。因此, 建议你从头到尾阅读这本书, 然后将个别章作为参考。但正如之前所说, 也可以不按顺序阅读个别章, 因为各章的项目是独立的。

有许多外部资源的链接。它们中的大多数是可选的, 并提供了关于该主题的额外详情。因此, 建议在电脑旁阅读本书, 以便在提到它们时可以打开链接。

有时你会看到带有奇怪缩进的代码，比如：

```
document.getElementById('end-of-time').play()  
}
```

这表示正在注解一大块代码并将其分解成块。这一块属于一个从位置 0 开始的更大的代码清单，这一小块代码不能自己运行。

其他时候，代码块没有缩进。这种情况下，可以假设代码片段是完整的：

```
ReactDOM.render(  
  <Content/>,  
  document.getElementById('content')  
)
```

如果看到一个美元符号(\$)，这表明是一条终端/命令提示符命令。例如：

```
$ npm install -g babel@5.8.34
```

在使用本书的时候，最重要的是要知道并记住：你必须乐在其中。如果不好玩，那就不是 JavaScript！

## 源代码

书中的所有代码可从 [www.manning.com/books/react-quickly](http://www.manning.com/books/react-quickly) 和 <https://github.com/azat-co/react-quickly> 下载，也可通过扫描封底二维码获得。请遵循文件夹命名约定 chNN，此处 NN 是以 0 开始的各章编号(例如，ch02 文件夹代表里面是第 2 章的代码)。GitHub 仓库中的源代码会不断更新，包括补丁、bug 修复，甚至可能是新的版本和风格(ES2020?)。访问网址 <http://reactquickly.co/demos>，可以得到各章的一些示例。

## 勘误

书中肯定会有打字错误。是的，虽然编辑们都很负责——他们都是 Manning 出版社的专业人士。但是，我仍对你能找到那个错误表示感激。请不要给我发送 bug 或打字错误。取而代之，你可以在本书的论坛(<https://forums.manning.com/forums/react-quickly>)上报告，或者在 <https://github.com/azat-co/react-quickly/issues> 上创建一个 GitHub issue。通过这种方式，其他人也可以受益于你的发现。

另外，请不要给我发邮件讨论技术问题或勘误。请把它们发在本书的论坛、GitHub 的页面(<https://github.com/azat-co/react-quickly>)或 Stack Overflow 上。其他人也许能比我更快(而且更好)地帮助你。

## 本书论坛

购买《快速上手 React 编程》后，可以免费访问由 Manning 出版社运营的私有 Web 论坛，在这里可以对本书发表评论，提出技术问题，并获得作者和其他用户的帮助。该论坛

的网址是 <https://forums.manning.com/forums/react-quickly>。也可以在 <https://forums.manning.com/forums/about> 上了解到更多关于 Manning 论坛和行为准则的信息。

Manning 出版社对读者的承诺是提供一个场所，在这里各位读者之间，读者与作者之间，可以碰撞出思想的火花。我们不承诺作者现身的次数，他们对该论坛的贡献是自愿的(并且是无酬劳的)。建议你尽量向作者问一些具有挑战性的问题，以激发他的兴趣！



# 目 录

## 第 I 部分 React 基础

第 1 章 初识 React .....	3
1.1 什么是 React .....	4
1.2 React 解决的问题 .....	5
1.3 使用 React 的好处 .....	6
1.3.1 简单性 .....	6
1.3.2 速度和可测试性 .....	11
1.3.3 生态和社区 .....	12
1.4 React 的缺点 .....	13
1.5 React 如何与 Web 应用集成 .....	13
1.5.1 React 类库和渲染目标 .....	14
1.5.2 单页面应用和 React .....	15
1.5.3 React 技术栈 .....	17
1.6 第一个 React 项目: Hello World .....	18
1.7 测验 .....	21
1.8 小结 .....	21
1.9 测验答案 .....	22
第 2 章 React 起步 .....	23
2.1 内嵌元素 .....	23
2.2 创建组件类 .....	26
2.3 属性 .....	29
2.4 测验 .....	34
2.5 小结 .....	34
2.6 测验答案 .....	34
第 3 章 JSX .....	35
3.1 JSX 是什么? 它有什么优点 .....	36
3.2 理解 JSX .....	38
3.2.1 使用 JSX 创建元素 .....	39
3.2.2 在组件中使用 JSX .....	40
3.2.3 在 JSX 中输出变量 .....	41
3.2.4 在 JSX 中使用属性 .....	42

3.2.5 创建 React 组件的方法 .....	46
3.2.6 JSX 中的 if/else .....	47
3.2.7 JSX 中的注释 .....	51
3.3 使用 Babel 设置 JSX 转译器 .....	51
3.4 React 和 JSX 陷阱 .....	55
3.4.1 特殊字符 .....	56
3.4.2 data-属性 .....	56
3.4.3 style 属性 .....	57
3.4.4 class 和 for .....	58
3.4.5 布尔类型的属性值 .....	58
3.5 测验 .....	59
3.6 小结 .....	59
3.7 测验答案 .....	59
第 4 章 与状态交互 .....	61
4.1 什么是 React 组件的状态 .....	62
4.2 使用状态 .....	63
4.2.1 访问状态 .....	63
4.2.2 设置初始状态 .....	65
4.2.3 更新状态 .....	67
4.3 状态和属性 .....	70
4.4 无状态组件 .....	71
4.5 有状态组件和无状态组件 .....	73
4.6 测验 .....	77
4.7 小结 .....	77
4.8 测验答案 .....	78
第 5 章 React 组件生命周期 .....	79
5.1 React 组件生命周期事件的 全景视图 .....	80
5.2 事件的分类 .....	80
5.3 实现生命周期事件 .....	82
5.4 执行所有事件 .....	84
5.5 挂载事件 .....	86
5.5.1 componentWillMount() .....	87

5.5.2	componentDidMount()	87	7.1.4	账户字段示例	132
5.6	更新事件	90	7.2	使用表单的其他方式	134
5.6.1	componentWillReceiveProps (newProps)	90	7.2.1	可捕获变更的非受控元素	135
5.6.2	shouldComponentUpdate()	91	7.2.2	不捕获变更的非受控元素	136
5.6.3	componentWillUpdate()	91	7.2.3	使用引用获取值	137
5.6.4	componentDidUpdate()	92	7.2.4	默认值	139
5.7	卸载事件	92	7.3	测验	140
5.8	一个简单示例	92	7.4	小结	141
5.9	测验	95	7.5	测验答案	141
5.10	小结	95	第 8 章	扩展 React 组件	143
5.11	测验答案	96	8.1	组件中的默认属性	144
第 6 章	React 事件处理	97	8.2	React 属性类型和验证	145
6.1	在 React 中处理 DOM 事件	97	8.3	渲染子组件	152
6.1.1	捕获和冒泡阶段	100	8.4	创建 React 高阶组件以实现 代码复用	154
6.1.2	React 事件的内幕	102	8.4.1	使用 displayName: 用以区分 父组件与子组件	156
6.1.3	使用 React SyntheticEvent 事件对象	105	8.4.2	使用扩展运算符: 传递所有 属性	157
6.1.4	使用事件和状态	108	8.4.3	使用高阶组件	158
6.1.5	传递事件处理程序和 属性	109	8.5	最佳实践: 展示组件与容器 组件	160
6.1.6	组件通信	112	8.6	测验	161
6.2	响应 React 不支持的 DOM 事件	113	8.7	小结	161
6.3	React 和其他库的集成: jQuery UI 事件	116	8.8	测验答案	162
6.3.1	集成按钮	116	第 9 章	项目: 菜单组件	163
6.3.2	集成标签	118	9.1	项目结构和脚手架	164
6.4	测验	119	9.2	不使用 JSX 构建菜单	165
6.5	小结	119	9.2.1	Menu 组件	165
6.6	测验答案	120	9.2.2	Link 组件	168
第 7 章	在 React 中使用表单	121	9.2.3	运行菜单组件	170
7.1	在 React 中使用表单的最佳 实践	121	9.3	在 JSX 中构建菜单	171
7.1.1	在 React 中定义表单及 响应事件	123	9.3.1	重构 Menu 组件	172
7.1.2	定义表单元素	125	9.3.2	重构 Link 组件	174
7.1.3	捕获表单变更	130	9.3.3	运行 JSX 项目	175
			9.4	测验	175
			9.5	小结	176



第 10 章 项目: Tooltip 组件	177
10.1 项目结构和脚手架	178
10.2 Tooltip 组件	179
10.2.1 toggle() 函数	180
10.2.2 render() 函数	181
10.3 运行 Tooltip 组件	183
10.4 测验	184
10.5 小结	184

第 11 章 项目: Timer 组件	185
11.1 项目结构和脚手架	186
11.2 应用架构	187
11.3 TimerWrapper 组件	189
11.4 Timer 组件	193
11.5 Button 组件	194
11.6 运行 Timer 组件	196
11.7 测验	196
11.8 小结	197

## 第 II 部分 React 架构

第 12 章 Webpack 构建工具	201
12.1 Webpack 的作用	201
12.2 添加 Webpack 到项目中	203
12.2.1 安装 Webpack 及其 依赖	204
12.2.2 配置 Webpack	205
12.3 模块化代码	207
12.4 运行 Webpack 并测试构建	208
12.5 热模块替换	210
12.5.1 配置 HMR	211
12.5.2 热模块替换实践	214
12.6 测验	216
12.7 小结	216
12.8 测验答案	216

第 13 章 React 路由	217
13.1 从零开始实现路由	218
13.1.1 建立项目	219
13.1.2 在 app.jsx 中创建路由 映射	220

13.1.3 在 router.jsx 中创建 Router 组件	221
13.2 React Router	222
13.2.1 React Router 的 JSX 样式	225
13.2.2 哈希记录	227
13.2.3 浏览器记录	227
13.2.4 使用 Webpack 安装 React Router 开发环境	228
13.2.5 创建布局组件	230
13.3 React Router 特性	233
13.3.1 使用 withRouter 高阶组件 访问路由器	234
13.3.2 以编程方式导航	235
13.3.3 URL 参数和其他路由 数据	235
13.3.4 在 React Router 中传递 属性	236
13.4 使用 Backbone 路由	237
13.5 测验	240
13.6 小结	241
13.7 测验答案	241

第 14 章 使用 Redux 处理数据	243
14.1 React 支持单向数据流	244
14.2 了解 Flux 数据体系结构	246
14.3 使用 Redux 数据类库	247
14.3.1 用 Redux 创建依照 Netflix 的应用	249
14.3.2 依赖和配置	250
14.3.3 启用 Redux	253
14.3.4 路由	253
14.3.5 合并 reducer	254
14.3.6 电影的 reducer	255
14.3.7 操作	258
14.3.8 操作创建器	259
14.3.9 将组件连接到数据 存储	260
14.3.10 分发操作	262



14.3.11	将操作创建器传递到 组件属性中 .....	263
14.3.12	运行 Netflix 的克隆版 .....	267
14.3.13	Redux 总结 .....	268
14.4	测验 .....	268
14.5	小结 .....	269
14.6	测验答案 .....	269
第 15 章	使用 GraphQL 处理数据 .....	271
15.1	GraphQL .....	272
15.2	给 Netflix 克隆版应用添加 服务器 .....	273
15.2.1	在服务器端安装 GraphQL .....	275
15.2.2	数据结构 .....	278
15.2.3	GraphQL 模式 .....	279
15.2.4	查询 API 并将响应保存 到数据存储 .....	281
15.2.5	显示电影列表 .....	285
15.2.6	GraphQL 总结 .....	287
15.3	测验 .....	287
15.4	小结 .....	288
15.5	测验答案 .....	288
第 16 章	使用 Jest 进行单元测试 .....	289
16.1	测试的类型 .....	290
16.2	为什么使用 Jest(对比 Mocha) .....	290
16.3	使用 Jest 进行单元测试 .....	291
16.3.1	在 Jest 中编写单元 测试 .....	293
16.3.2	Jest 断言 .....	294
16.4	使用 Jest 和 TestUtils 进行 React UI 测试 .....	296
16.4.1	使用 TestUtils 查找 元素 .....	298
16.4.2	UI 测试密码部件 .....	299
16.4.3	浅渲染 .....	303
16.5	TestUtils 总结 .....	305
16.6	测验 .....	305
16.7	小结 .....	305
16.8	测验答案 .....	306
第 17 章	在 Node 中使用 React 和 同构 JavaScript .....	307
17.1	为什么在服务器端使用 React? 什么是同构 JavaScript? .....	308
17.1.1	正确的页面索引 .....	308
17.1.2	更快的加载速度、更好的 性能 .....	309
17.1.3	更好的代码可维护性 .....	310
17.1.4	在 React 和 Node 中使用 同构 JavaScript .....	310
17.2	在 Node 上使用 React .....	312
17.3	React 和 Express: 在服务器端 渲染组件 .....	314
17.3.1	在服务器端渲染简单的 文本 .....	315
17.3.2	渲染 HTML 页面 .....	316
17.4	使用 Express 和 React 的同构 JavaScript .....	322
17.4.1	项目目录结构和配置 .....	324
17.4.2	启动服务器 .....	325
17.4.3	使用 Handlebars 的服务器 端布局模板 .....	329
17.4.4	在服务器上编写 React 组件 .....	332
17.4.5	客户端 React 代码 .....	333
17.4.6	配置 Webpack .....	334
17.4.7	运行应用 .....	336
17.5	测验 .....	340
17.6	小结 .....	340
17.7	测验答案 .....	340
第 18 章	使用 React Router 创建一个 网上书店 .....	341
18.1	项目结构和 Webpack 配置 .....	343
18.2	HTML 主页 .....	346
18.3	创建组件 .....	347
18.3.1	主文件: app.jsx .....	347

18.3.2	Cart 组件 .....	353	20.2	实现 Web 服务器 .....	385
18.3.3	Checkout 组件 .....	355	20.2.1	定义 RESTful API .....	386
18.3.4	Modal 组件 .....	356	20.2.2	在服务器端渲染 React .....	387
18.3.5	Product 组件 .....	357	20.3	添加浏览器脚本 .....	387
18.4	启动项目 .....	359	20.4	创建服务器端模板 .....	388
18.5	测验 .....	359	20.5	实现 Autocomplete 组件 .....	389
18.6	小结 .....	359	20.5.1	Autocomplete 组件的 测试 .....	389
第 19 章	使用 Jest 测试密码 .....	361	20.5.2	Autocomplete 组件的 代码 .....	390
19.1	项目结构和 Webpack 配置 .....	362	20.6	整合 .....	393
19.2	HTML 主页 .....	365	20.7	测验 .....	395
19.3	实现强密码模块 .....	366	20.8	小结 .....	396
19.3.1	测试 .....	366	附录 A	安装本书相关应用 .....	397
19.3.2	代码 .....	367	附录 B	React 速查表 .....	405
19.4	实现 Password 组件 .....	369	附录 C	Express 速查表 .....	413
19.4.1	测试 .....	369	附录 D	MongoDB 和 Mongoose 速查表 .....	419
19.4.2	代码 .....	370	附录 E	ES6 简介 .....	423
19.5	实践 .....	375			
19.6	测验 .....	376			
19.7	小结 .....	377			
第 20 章	使用 Jest、Express 和 MongoDB 实现自动完成 .....	379			
20.1	项目结构和 Webpack 配置 .....	381			

# 第 I 部分

## React 基础

你好！我是 Azat Mardan，我即将带你进入一段美妙的 React 世界之旅。这将使前端开发更加愉悦，代码更易于编写和维护，用户也会对应用的速度感到欣喜。React 改变了 Web 开发的游戏规则：React 社区开创了许多方法、术语和设计模式，并且其他库也都追随了 React 的脚步。

我已经在线上直播和个人研讨会上教授这些内容 20 余次，听众是数以百计的软件工程师，他们具有不同的背景和资历。因此，这些内容是经过实践检验过的：它们是我的 React 基础课程经过提炼后的、最有效的书面版本。这些章对于你熟悉 React 术语至关重要。

第 1 至第 11 章是多人近两年来工作的成果，里面介绍的主题层层递进，需要快速阅读。阅读这些章的最佳方法是从第 1 章开始，按照顺序进行。每章都包含一段视频，第 1 至第 8 章的最后还有一个测验，第 9 至第 11 章是具体项目，包含需要你自主完成开发的作业。

总而言之，本书的这一部分为 React 概念、模式和特性构建了坚实的基础。到了国外，不通过学习就可以理解它们的语言吗？显然不行，这就是为什么在尝试构建复杂应用之前，必须先学习 React “语言” 的原因。因此，学习这些 React 基本概念至关重要。学习 React 语言正是接下来的 11 章要完成的工作。

让我们开始使用 React，并学会流利的 React 语言吧！



# 第 1 章

## 初识 React

本章内容：

- 理解什么是 React
- 使用 React 解决问题
- 把 React 添加到 Web 应用中
- 编写第一个 React 应用：Hello World

2000 年初，在我开始从事 Web 开发工作时，所需要的只是一些 HTML 和诸如 Perl 或 PHP 这样的服务器端语言。那些使用 `alert()` 弹出信息的美好时光只是为了调试前端代码。事实上，随着互联网的发展，构建网站的复杂度已经急剧增加。网站已经成为具有复杂用户界面、业务逻辑和数据层的 Web 应用，并且需要随时进行修改和更新，且通常是实时的。

许多 JavaScript 模板库已被创建用来尝试解决处理复杂用户界面(UI)的问题。但是，它们仍需要开发人员坚持旧的关注点分离原则——分离样式(CSS)、数据和结构(HTML)以及动态交互(Javascript)，而它们并不能满足现代需求。(还记得术语 DHTML 吗？)

相比之下，React 提供了一种简化前端开发的新方法。React 是一个功能强大的 UI 库，提供了许多大公司(如 Facebook、Netflix 和 Airbnb)已经采纳和认定的替代方法。React 允许使用 JavaScript 创建可重用的 UI 组件，而不是为 UI 定义一次性的模板，可以在网站中反复使用这些组件。

还需要验证码控制器或日期选择器吗？使用 React 定义一个 `<Captcha>` 或 `<DatePicker>` 组件，就可将它们添加到表单中：这就是一个简单的插件组件，具备与后端通信的所有功能和逻辑。当用户输入四个或更多个字母时，是否需要一个自动完成框来异步查询数据库？定义一个 `<Autocomplete charNum="4" />` 组件就可以进行异步查询。还可以选择是否存在一个文本框 UI 或者没有 UI，此时可以使用另一个自定义表单元素，也许是 `<Autocomplete textbox="..." />`。

这种做法并不新奇。“创建可组合的 UI”的方法已经存在很长时间了，但 React 是第一个使用无模板的纯 JavaScript 来实现的。这种方法已被证明易于维护、重用和扩展。

React 是一个很好的 UI 库，它应该成为 Web 前端工具包的一部分；但它并不是所有 Web 前端开发的完整解决方案。在本章中，我们将介绍在应用中使用 React 的优缺点，以及如何将其适配到现有的 Web 开发技术栈中。

本书的第 I 部分重点介绍 React 的主要概念和特性，第 II 部分着重讨论与 React 相关的库，以构建更复杂的前端应用(也就是 React 技术栈或 React 全家桶)。每部分都展示使用 React 和最受欢迎的库进行的新系统(绿地)和遗留系统(棕地)的开发<sup>1</sup>，所以你可以了解如何在真实场景中使用它。

### 各章的视频和源代码

每个人的学习方法都不同。有些人喜欢文字和视频，有些人更喜欢通过面授教学学习。本书的每一章都包含一段简短的视频，在不到 5 分钟的时间内就可以解释该章的要点。观看视频是可选的，如果喜欢视频方式或需要复习，它们可以为你提供摘要。观看每段视频后，可以决定是否需要阅读该章，或是跳到下一章。

本章示例的源代码位于 [www.manning.com/books/reactquickly](http://www.manning.com/books/reactquickly) 和 [https://github.com/azat-co/react-quickly](https://github.com/azat-co/react-quickly/tree/master/ch01) (在 Github 仓库 <https://github.com/azat-co/react-quickly> 的 ch01 文件夹中)。可以在 <http://reactquickly.co/demos> 上找到一些示例。

## 1.1 什么是 React

要合理解释 React，首先需要给它下个定义。那么，什么是 React 呢？它是一个 UI 组件库。这些 UI 组件通过 React 使用 JavaScript 而不是一种特殊的模板语言创建。这种方法被称为“创建可组合的 UI”，它是 React 的理念基础。

React UI 组件是高度自包含、有特定关注点的功能单元。例如：可能有日期选择器、验证码、地址和邮政编码组件。这些组件具有视觉展示和动态逻辑。某些组件甚至可以自己与服务器通信：例如，自动完成组件可能会从服务器提取自动完成列表。

### 用户界面

广义上，用户界面<sup>2</sup>是促进计算机和人类之间交互的一切。想想一张穿孔卡片(早期的为计算机输入指令用的一种纸质卡片，这个术语源自 Herman Hollerith，他在 1890 年为美国人口局设计了电子制表系统，采用了穿孔卡片)或鼠标，它们都是 UI。当进入软件时代时，工程师们谈论的图形用户界面(Graphical User Interface, GUI)是早期的个人电脑(如 Mac 和 PC)的先驱。GUI 包括菜单、文本、图标、图片、边框和其他元素。Web 元素只是 GUI 的一小部分，它们位于浏览器中，但 Windows、OS X 和其他操作系统中也有用于桌面应用的元素。当在本书中提到 UI 时，意指 Web GUI。

<sup>1</sup> “棕地”指的是包含遗留代码的已有系统，“绿地”指的是没有遗留代码的新系统，参阅 [https://en.wikipedia.org/wiki/Brownfield\\_\(software\\_development\)](https://en.wikipedia.org/wiki/Brownfield_(software_development))

<sup>2</sup> [https://en.wikipedia.org/wiki/User\\_interface](https://en.wikipedia.org/wiki/User_interface)



不要将基于组件的架构(Component-Based Architecture, CBA)和 Web 组件搞混淆, Web 组件只是在 React 之前存在的最新 CBA 实现之一。这种架构通常被认为比单个庞大的 UI 更容易重用、维护和扩展。React 带来的好处是可以使用纯 JavaScript(无模板)和一种新的视角来看待组件的组合。

## 1.2 React 解决的问题

React 解决了什么问题?放眼近几年的 Web 开发,你会注意到在前端应用中构建和管理复杂 Web UI 存在的问题,React 主要就是为了解决这些问题而生。试想一下,像 Facebook 这样的大型网络应用,开发此类应用时最痛苦的任务之一就是管理视图对数据变更的响应。

浏览 React 官网以便了解这个问题的更多信息:“我们构建 React 来解决如下问题:构建包含随时间变化的数据的大型应用”<sup>3</sup>。这很有趣!我们还可以查看 React 的历史来了解更多信息。React Podcast<sup>4</sup>上的一个讨论提到,React 的创始人 Jordan Walke 正在 Facebook 上解决一个问题:一个自动完成字段,有多个数据源对其进行更新,这些数据都是从后端异步获取的。确定在哪里插入新行以便重用 DOM 元素变得越来越复杂。Walke 决定每次重新生成这个字段的 UI 展示(DOM 元素)。这个解决方案因其简洁性而变得非常优雅:UI 作为函数,数据作为参数并调用这种函数,并且使渲染视图结果变得可预测。

之后,事实证明在内存中生成元素非常快,实际瓶颈出现在渲染到 DOM 中时。但是 React 团队提出了一种可以避免非必要 DOM 渲染的算法。这使得 React 非常快速(在性能方面)。React 的卓越性能表现,以及对开发人员友好的基于组件的架构是非常成功的组合。React 的这些相关内容和其他优点将在下一节中介绍。

React 解决了 Facebook 的初始问题,许多大公司都认同这种做法。React 采用的方案非常稳固,而且人气逐月攀升。React 源自 Facebook<sup>5</sup>,但现在不仅被 Facebook 使用,Instagram、PayPal、Uber、Sberbank、Asana<sup>6</sup>、Khan Academy<sup>7</sup>、HipChat<sup>8</sup>、Flipboard<sup>9</sup>以及 Atom<sup>10</sup>都在使用 React。<sup>11</sup>大多数这些应用最初使用的是其他技术(通常是带有模板引擎的 Angular 或 Backbone),但现在都非常乐于切换到 React。

---

3 React官网,“Why React?”, 2016/03/24, <http://bit.ly/2mdCJKM>

4 React Podcast, “8. React, GraphQL, Immutable & Bow-Ties with Special Guest Lee Byron”, 2015/12/31, <http://mng.bz/W1X6>

5 “Introduction to React.js”, 2013/07/08, <http://mng.bz/86XF>

6 Malcolm Handley和Phips Peter, “Why Asana Is Switching to TypeScript”, Asana Blog, 2014/11/14, <http://mng.bz/zXKo>

7 Joel Burget, “Backbone to React”, <http://mng.bz/WGEQ>

8 Rich Manalang, “Rebuilding HipChat with React.js” *Atlassian Developer*, 2015/02/10, <http://mng.bz/r0w6>

9 Michael Johnston, “60 FPS on the Mobile Web” Flipboard, 2015/02/10, <http://mng.bz/N5F0>

10 Nathan Sobo, “Moving Atom to React” Atom, 2014/07/02, <http://mng.bz/K94N>

11 另请参阅JavaScript使用统计信息, <http://libscore.com/#React>



## 1.3 使用 React 的好处

所有新的库或框架都会声称相比它的前辈在某些方面做得更好。一开始，我们拥有了 jQuery，同使用原生 JavaScript 编写跨浏览器代码相比，jQuery 是一次跳跃式的提升。你应该还记得，一个 AJAX 调用需要写多行代码，而且还必须考虑兼容 IE 浏览器和类 WebKit 浏览器。使用 jQuery，这只需要一个调用：例如 `$.ajax()`。过去，jQuery 被称为一个框架，可如今它已经不再是了！现在，框架是一个更大且更强的概念。

与 Backbone 和 Angular 类似，每个新生代 JavaScript 框架都会带来很多新的优点。在这一点上 React 并非唯一。React 带来的革新是它挑战了大多数流行的前端框架所使用的一些核心概念：例如需要拥有模板这样的想法。

以下列表突出显示了 React 与其他库和框架相比的一些优点：

- 更简单的应用——React 使用纯 JavaScript 构建 CBA；声明式风格；强大且开发者友好的 DOM 抽象(不只是 DOM，还有 iOS、Android 等)。
- 快速的 UI——React 通过虚拟 DOM 和智能协调算法提供出色的性能，借助此优点，可以让你的性能测试不再需要启动一个无界面浏览器。
- 编写更少的代码——React 伟大的社区和丰富的组件生态为开发人员提供了各种库和组件。考虑框架选型时，这一点至关重要。

许多特点使得 React 相比其他大多数前端框架更容易使用。让我们逐个打开这些项目，从它的简单性开始。

### 1.3.1 简单性

计算机科学中的简单性概念受到开发人员 and 用户的高度认可。它不同于易用性。简单的事情可能实现起来会很困难，但最终会更加优雅和高效。通常，一件容易的事情最终会变得复杂。简单性与 KISS 原则(保持简单、傻瓜)<sup>12</sup>密切相关。要点是：越简单的系统工作得越好。

React 通过为软件工程师提供更好的 Web 开发体验，提供了一个更简单的解决方案。当我开始使用 React 时，经历了一次相当大的转变，我需要时刻提醒自己使用纯粹的、无框架的 JavaScript 而不是 jQuery。

在 React 中，这种简单性通过以下特性实现：

- 命令式风格之上的声明式风格——React 通过自动更新视图来实现在命令式风格之上拥抱声明式风格。
- 使用纯 JavaScript 的基于组件的架构——React 不会为其组件使用领域特定语言(Domain Specific Language, DSL)，只是纯 JavaScript。在实现同一功能时没有分离。
- 强大的抽象——React 拥有与 DOM 进行交互的简化方法，允许进行规范化的事件处理和使用其他接口，就像在浏览器中一样。

让我们逐一介绍。

---

12 [https://en.wikipedia.org/wiki/KISS\\_principle](https://en.wikipedia.org/wiki/KISS_principle)

### 命令式风格之上的声明式风格

首先, React 在命令式风格之上拥抱了声明式风格。声明式风格意味着开发人员编写代码以示事物应该是什么,而不是一步步(命令式)地去做什么。但为什么声明式风格是更好的选择呢?因为声明式风格降低了复杂性,使代码更容易阅读和理解。

考虑这个简短的 JavaScript 示例,其中说明了声明式和命令式编程之间的区别。假设需要创建一个数组(arr2),它的元素是将另一个数组(arr)的元素加倍之后的结果。可以使用 for 循环遍历数组,并告诉系统将元素乘以 2 并创建一个新元素(arr2[i]=):

```
var arr = [1, 2, 3, 4, 5],
    arr2 = []
for (var i=0; i<arr.length; i++) {
  arr2[i] = arr[i]*2
}
console.log('a', arr2)
```

这段代码的运行结果是将每个元素乘以 2 并按如下所示将结果输出到控制台上:

```
a [2, 4, 6, 8, 10]
```

这个例子阐述了命令式编程,它可以正常运行。但由于代码的复杂性,它有可能会变得无法正常运行。当有太多的命令式语句时,会变得难以理解最终的结果应该是什么。幸运的是,可以通过 map() 使用声明式写法重写这段逻辑:

```
var arr = [1, 2, 3, 4, 5],
    arr2 = arr.map(function(v, i){ return v*2 })
console.log('b', arr2)
```

输出为 b[2,4,6,8,10]。变量 arr2 与前面的例子相同。哪段代码更容易阅读和理解?我个人陋见是:采用声明式写法的那段。

以下命令式代码从一个对象中获取嵌套的值。表达式需要返回一个基于字符串(如 account 或 account.number)的值,使得这些语句输出 true:

```
var profile = {account: '47574416'}
var profileDeep = {account: { number: 47574416 }}
console.log(getNestedValueImperatively(profile, 'account') === '47574416')
console.log(getNestedValueImperatively(profileDeep, 'account.number')
  === 47574416)
```

命令式风格的代码从字面上告诉系统要做什么来获取你需要的结果:

```
var getNestedValueImperatively = function getNestedValueImperatively
  (object, propertyName) {
  var currentObject = object
  var propertyNamesList = propertyName.split('.')
  var maxNestedLevel = propertyNamesList.length
  var currentNestedLevel

  for (currentNestedLevel = 0; currentNestedLevel < maxNestedLevel;
    currentNestedLevel++) {
```



```
if (!currentObject || typeof currentObject === 'undefined')
  return undefined
currentObject = currentObject[propertyNamesList[currentNestedLevel]]
}
return currentObject
}
```

与声明式风格相比(关注结果), 局部变量的数量减少了, 从而简化了逻辑:

```
var getValue = function getValue(object, propertyName) {
  return typeof object === 'undefined' ? undefined : object[propertyName]
}

var getNestedValueDeclaratively = function getNestedValueDeclaratively(object,
  propertyName) {
  return propertyName.split('.').reduce(getValue, object)
}

console.log(getNestedValueDeclaratively({bar: 'baz'}, 'bar') === 'baz')
console.log(getNestedValueDeclaratively({bar: { baz: 1 }}, 'bar.baz') === 1)
```

大多数程序员都受过命令式风格的编码训练, 但通常声明式代码更加简单。在这个例子中, 使用较少的变量和语句可以使声明式代码更容易掌握。

那只是一些 JavaScript 代码。和 React 有什么关系呢? 当组合 UI 时, React 采用相同的声明式风格。首先, React 开发人员以声明式风格描述 UI 元素。然后, 当对这些 UI 元素生成的视图进行修改时, React 会自己完成更新。这太棒了!

当需要让视图发生变更时, React 声明式风格的便利性将被完全展现出来。这些变更被称为内部状态的变更。当变更发生时, React 会自动更新视图。

**注意:** 第 4 章会介绍状态的工作原理。

在底层, React 使用虚拟 DOM 来查找浏览器中已有视图和新视图之间的差异(增量)。这个过程称为 DOM diff, 又称为状态和视图的协调(使它们保持相似性)。这意味着开发人员不需要担心显式地修改视图, 他们需要做的就是更新状态, 视图会根据需要自动更新。

相反, 使用 jQuery 则需要强制执行更新。通过操作 DOM, 开发人员可以通过编程修改网页或网页的一部分(更可能的情况), 而无须重新渲染页面。调用 jQuery 方法时所做的就是进行 DOM 操作。

一些框架, 比如 Angular, 可以执行自动的视图更新。在 Angular 中, 这被称为双向数据绑定, 这基本上意味着视图和模型之间有双向的数据通信/同步。

jQuery 和 Angular 方式不是很好, 有两个原因。可以把它们想象成两个极端: 其中一个极端, jQuery 没有做任何事情, 开发人员需要手动实现所有的更新; 另一个极端, 框架(Angular)正在做一切事情。

jQuery 方式很容易出错, 需要做更多的工作来实现。此外, 直接操作常规 DOM 的方式可以让简单的 UI 正常工作, 但是当处理 DOM 树中的大量元素时, 这是有限制的。情况就是这样, 因为命令式函数比声明式语句更难以看到结果。

Angular 方式很难理解, 因为双向绑定, 事情可能会迅速失控。当加入越来越多的逻辑



辑时，可能突然间，不同的视图在更新模型，这些模型又更新了其他视图。

Angular 方式固然比命令式 jQuery 更加可读，而且需要手动编码的量也更少！但仍存在另一个问题。Angular 依赖使用 ng 指令(例如 ng-if)的模板和 DSL。下一节将讨论 Angular 方式的缺点。

### 使用纯 JavaScript 的基于组件的架构

基于组件的架构<sup>13</sup>在 React 之前就已存在。分离关注点、松散耦合和代码重用是这种架构的核心，因为它提供了许多好处。软件工程师(包括 Web 开发者)很喜欢 CBA。React 中的 CBA 构建单元是 Component 类。与其他 CBA 架构一样，它有许多好处，代码重用是主要的(可以编写更少的代码)。

在 React 之前，一直缺少此类架构的纯 JavaScript 实现。当使用 Angular、Backbone、Ember 或其他类 MVC 前端框架时，会有一个文件用于 JavaScript，另一个文件用于模板(Angular 对组件使用“指令”这个术语)。对于单个组件，存在两种语言(和两个或更多个文件)会有一些问题。

当需要在服务器端渲染 HTML 时，HTML 和 JavaScript 分离可以工作得很好，JavaScript 的作用仅仅是让文本闪烁起来。现在单页面应用(SPA)需要处理复杂的用户输入并在浏览器上执行渲染。这意味着 HTML 和 JavaScript 在功能上是紧密耦合的。对于开发人员而言，如果开发一个项目(组件)时，不再需要将 HTML 和 JavaScript 分离，这会更有意义。

来看下面这段 Angular 代码，它根据 user.session 的值来显示不同的链接：

```
<a ng-if="user.session" href="/logout">Logout</a>
<a ng-if="!user.session" href="/login">Login</a>
```

虽然可以读懂，但可能会疑惑 ng-if 需要什么类型的值：布尔值抑或字符串。会隐藏元素还是仅仅不渲染？在 Angular 案例中，无法确定元素在值为 true 或 false 时是否将被隐藏，除非熟悉这个特定 ng-if 指令的工作原理。

将上述代码与下面使用 if/else 实现条件渲染的 React 代码进行比较。user.session 的值应该是什么，并且如果 user.session 的值为 true，哪个元素会被渲染(logout 或 login)，对此毫无疑问。为什么？因为这只是 JavaScript：

```
if (user.session) return React.createElement('a', {href: '/logout'}, 'Logout')
else return React.createElement('a', {href: '/login'}, 'Login')
```

当需要遍历数组并打印属性时，模板很有用。我们一直在和数据列表打交道，下面来看看 Angular 中的 for 循环。如前所述，在 Angular 中，需要使用带有指令的 DSL，for 循环的指令是 ng-repeat：

```
<div ng-repeat="account in accounts">
  {{account.name}}
</div>
```

模板存在的一个问题是开发人员经常需要学习另外一种语言。在 React 中，使用的是纯 JavaScript，这意味着不需要学习新语言！下面是使用纯 JavaScript 编写账户名称列表的

13 <http://mng.bz/a65r>

UI 示例:

```
accounts.map(function(account) { ←—— 将迭代器表达式作为参数的常规 JavaScript 方法14
  return React.createElement('div', null, account.name)
})                                ←—— 迭代器表达式返回一个使用<div>标签包裹的账户名称
```

想象一下, 假设正在对账户列表进行一些修改。需要显示账户数量和其他字段。除了 name 之外, 如何才能知道账户中还有哪些其他字段呢?

需要打开调用和使用该模板的 JavaScript 文件, 然后找到 accounts 对象以查看其属性。因此, 模板存在的第二个问题是: 数据的逻辑和如何呈现数据的描述是分开的。

将 JavaScript 和标记放在一起会更好, 因为不必在文件和语言之间切换, 这正是 React 的工作原理。你将在 Hello World 示例中看到 React 如何快速渲染元素。

注意: 分离关注点通常会是一种很好的模式。简言之, 它意味着分离不同的功能, 如数据服务、视图层等。当使用模板标记和对应的 JavaScript 代码时, 是在处理同一个功能问题, 这就是为什么虽有两个文件(.js 和.html)却不是关注点分离的原因。

现在, 如果要在渲染列表中显式地设置跟踪条目的方法(例如, 确保列表中没有重复项), 可以使用 Angular 的 track by 特性:

```
<div ng-repeat="account in accounts track by account._id">
  {{account.name}}
</div>
```

如果想跟踪数组的下标, 可以使用 \$index:

```
<div ng-repeat="account in accounts track by $index">
  {{account.name}}
</div>
```

但是, 我和其他许多开发者关心的是, \$index 背后的魔法到底是什么? 在 React 中, 可以使用 map() 的参数作为 key 属性的值:

```
accounts.map(function(account, index) { ←—— 使用 Array.map() 提供的数组元素值(account)
  return React.createElement('div', {key: index}, account.name) ←——
})                                返回一个 React 元素<div />, 里面包含一个值为 index
                                的 key 属性, 并且设置内部文本为 account.name
```

值得注意的是: map() 不是 React 独有的, 可以将它与其他框架结合使用, 因为它是 JavaScript 语言的一部分。但是 map() 天生的声明式特点使它和 React 成为完美组合。

我没有选择 Angular, 虽然它是一个很好的框架。如果一个框架使用 DSL, 那么势必需要学习它的魔法变量和方法, 这超出了我的底线。而在 React 中, 可以使用纯 JavaScript。

如果选择 React, 那么即使不在 React 中, 也可以将所学知识应用到下一个项目中。另一方面, 如果使用 X 模板引擎(或 Y 框架内置的 DSL 模板引擎), 你将被锁定在该系统中,

<sup>14</sup> <http://mng.bz/555J>



并将自己描述成 X/Y 开发人员。你所学的知识无法在不使用 X/Y 的项目中应用。总而言之，基于纯 JavaScript 组件的架构关注的是使用独立、封装良好的可重用组件，从而可以确保实现更好的关注点分离，而不需要 DSL、模板或指令。

在与许多开发团队合作后，我观察到与简单性相关的另一个因素。与 MVC 框架(React 不是 MVC 框架，所以我会停止这种比较)和具有特殊语法的模板引擎(例如 Angular 指令和 Jade/Pug)相比，React 有一条更好、更浅、更渐进式的学习曲线。原因在于，大多数模板引擎使用自己的 DSL 来建立抽象，使用类似 if 条件和 for 循环的方式而不是使用 JavaScript 重塑这些事情。

### 强大的抽象

React 拥有强大的文档模型抽象功能，换言之，它隐藏了底层接口并提供归一化/合成的方法和属性。例如，当在 React 中创建 onClick 事件时，事件处理程序将不会收到浏览器原生的特定事件对象，而是收到封装了原生事件对象的合成事件对象。无论在什么浏览器中运行代码，都可以期待合成事件具有相同的行为。React 还有一组触摸事件的合成事件，这对于构建移动设备的 Web 应用非常有用。

React DOM 抽象的另一个例子是可以在服务器端渲染 React 元素，这可以方便搜索引擎优化(SEO)和/或提高性能。

相比渲染服务器后端返回的 HTML 字符串和 DOM，使用 React 组件进行渲染时有更多的选择。我们将在 1.5.1 节中介绍。并且关于 DOM，React 最受追捧的优点之一就是其卓越的性能表现。

### 1.3.2 速度和可测试性

除了必要的 DOM 更新，你的框架可能会执行一些不必要的更新，这使得复杂 UI 的性能变得更糟。当网页上有很多动态 UI 元素时，用户会受较大影响且很痛苦。

另一方面，React 的虚拟 DOM 只存在于 JavaScript 内存中。每次有数据更改时，React 首先使用虚拟 DOM 比较差异，只有当 React 知道渲染将会有更改时才会更新实际的 DOM。图 1.1 展示了有数据更改时对 React 虚拟 DOM 工作原理的高级概述。

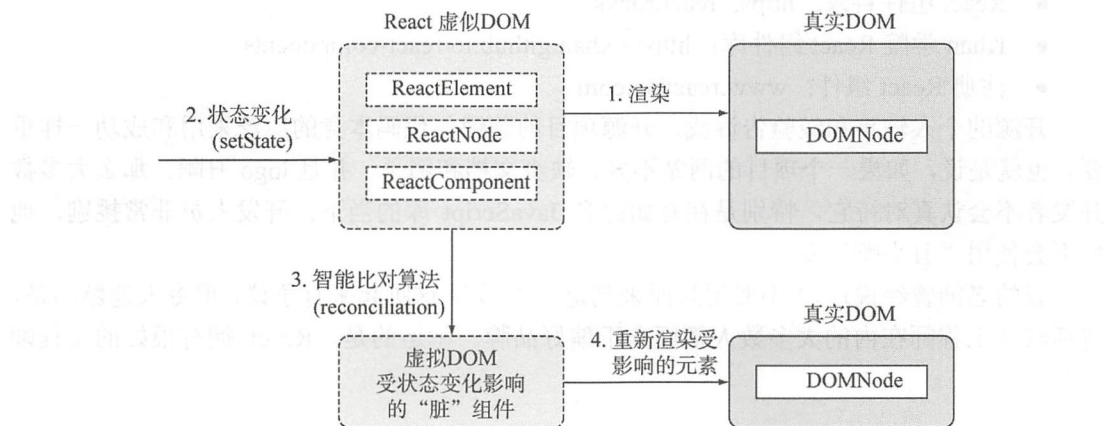


图 1.1 一旦组件被渲染，如果它的状态发生变化，就与内存中的虚拟 DOM 进行比较，在需要的时候重新进行渲染



最终, React 仅更新绝对必要的部分, 以使内部状态(虚拟 DOM)和视图(真实 DOM)保持一致。例如, 一个<p>元素, 使用组件状态为其添加文本, 则只会更新文本(即 innerHTML), 而不是更新元素本身。与渲染整个元素集合(甚至整个页面(服务器端渲染))相比, 这样做会提高性能。

注意: 如果喜欢挑战算法和算法复杂度, 有两篇好文解释了 React 团队如何设法将  $O(n^3)$  问题变成  $O(n)$  问题: “Reconciliation”, 参考 React 官网(<http://mng.bz/PQ9X>); 以及 Christopher Chedeau 所写的 “React’s Diff Algorithm”, 参考 <http://mng.bz/68L4>。

虚拟 DOM 的另一个好处就是可以脱离诸如 PhantomJS(<http://phantomjs.org>)这样的无界面浏览器进行单元测试。有一个名为 Jest(<https://facebook.github.io/jest>)的 Jasmine (<http://jasmine.github.io>)层, 可以让你在命令行上测试 React 组件。

### 1.3.3 生态和社区

最后, 但并非最不重要的一点是: React 受 Facebook 开发人员以及 Instagram 的同行们支持, 与 Angular 和其他库一样, 背后拥有一家大公司提供健全的测试基础(已被部署到数百万浏览器中)、对未来提供保证以及对贡献速度的提升。

React 社区让人难以置信, 大多数情况下, 开发人员甚至不必自己实现大量的代码。看看这些社区资源:

- React 组件清单: <https://github.com/brillout/awesome-react-components> 和 <http://devarchy.com/react-components>
- 一套实现 Google Material Design 规范(<https://design.google.com>)的 React 组件库: <http://react-toolbox.com>
- Material Design React 组件库: [www.material-ui.com](http://www.material-ui.com)
- 遵循 Office 设计语言以及 Office 和 Office 360 体验的 React 组件集合: <https://github.com/OfficeDev/office-ui-fabric-react>
- 开源 JS(大部分是 React)软件包的意见目录: <https://js.coach>
- React 组件目录: <https://react.rocks>
- Khan 学院 React 组件库: <https://khan.github.io/react-components>
- 注册 React 组件: [www.reactjsx.com](http://www.reactjsx.com)

开源的个人轶事和经验告诉我, 开源项目的营销与代码本身的广泛采用和成功一样重要。也就是说, 如果一个项目的网站不好, 缺少文档和例子, 并且 logo 丑陋, 那么大多数开发者不会认真对待它, 特别是在有如此多 JavaScript 库的当下。开发人员非常挑剔, 他们不会使用“丑小鸭”库。

我的老师曾经说过, “不要用封面来判定一本书”, 这听起来有争议, 但令人遗憾的是, 包括软件工程师在内的大多数人都倾向于偏好品牌。幸运的是, React 拥有很好的工程师知名度。

## 1.4 React 的缺点

当然，一切事物都有缺点，React 也不例外，但完整的缺点列表取决于询问的对象。不同的人会有一些差异，比如声明式和命令式是非常主观的。因此，它们既可为利，亦可为弊。下面是我的 React 缺点清单(如同任何类似的清单，这可能有偏见，因为这是基于从其他开发人员那里听到的意见)：

- React 不是一个完备的、瑞士军刀型框架。开发人员需要将其与 Redux 或 React Router 等库配合使用，以实现与 Angular 或 Ember 相当的功能。如果需要一个简约的 UI 库来与现有技术栈集成，这也可以成为优势。
- React 不如其他框架那么成熟，它的核心 API 仍然在变化，尽管 0.14 版本之后变化很少。React 的最佳实践(以及组件、插件和附加组件的生态系统)仍然在持续发展。
- React 使用一种新的 Web 开发方法，JSX 和 Flux(通常用作 React 的数据管理类库)可能会对初学者有一定的门槛。缺乏可用于掌握 React 的最佳实践、好书、课程和资源。
- React 只有单向绑定。虽然单向绑定对于复杂应用更好，并且消除了大量的复杂性，但是一些开发人员(特别是 Angular 开发人员)已习惯双向绑定，他们会发现自己需要编写更多的代码。我将在第 14 章比较 React 的单向绑定和 Angular 的双向绑定，包括数据处理。
- React 不是开箱即用的响应式库(比如在响应式编程和架构中，它们更具备事件驱动、弹性和响应性的特点)。开发人员需要使用其他工具(例如 Reactive Extensions (RxJS, <https://github.com/Reactive-Extensions/RxJS>))来构建支持 Observables 的异步数据流。

继续上述对 React 的介绍，我们来看看如何将之集成到一个 Web 应用中。

## 1.5 React 如何与 Web 应用集成

某种程度上，React 本身没有 React Router 或数据处理库，和框架(如 Backbone、Ember 和 Angular)的可比性更小，更适合和处理 UI 的库进行比较，例如模板引擎(Handlebars、Blaze)和 DOM 操作类库(jQuery、Zepto)。实际上，许多团队已经将传统的模板引擎(如 Backbone 的 Underscore 或 Meteor 的 Blaze)切换到 React，并取得非常大的成功。例如，Paypal 从 Dust 转为 React，与本章前面列出的许多其他公司一样。

可以针对部分 UI 使用 React。例如，假设有一个使用 jQuery 构建的贷款申请表页面。可以逐渐地将这个前端应用转换为采用 React，首先把城市和州名字段转换为根据邮政编码自动填充。表单其余部分可以继续使用 jQuery。然后，如果要继续，可将其余表单元素从 jQuery 转换为 React，直到整个页面都基于 React。采用类似的方法，许多团队将 React 与 Backbone、Angular 或其他现有的前端框架成功地整合在了一起。

对前端开发而言，React 不关注后端技术选型。换言之，无须依赖 Node.js 后端或 MERN(MongoDB、Express、React 和 Node.js)来使用 React。React 和其他后端技术(如



Java、Ruby、Go、Python)配合得同样很好。毕竟 React 只是一个 UI 库。可以将其与任何后端和前端数据类库(Backbone、Angular、Meteor)集成在一起。

总结一下 React 如何与 Web 应用集成, 最常见于下列情况:

- 作为 React 单页面技术栈中的 UI 库, 例如 React+React 和 Router+Redux。
- 作为非完整 React 单页面技术栈中的 UI 库(MVC 中的 V), 例如 React+Backbone。
- 作为任何前端技术栈中的 UI 组件, 例如 jQuery+服务器端渲染技术栈中的 React 自动完成输入组件。
- 作为传统的富服务器端 Web 应用、混合 Web 应用或同构 Web 应用的服务器端模板库。
- 作为移动应用中的 UI 库, 例如 React Native iOS 应用。
- 作为不同渲染目标的 UI 描述库(在下一节中讨论)。

React 与其他前端技术一起配合得很好, 但它主要用作单页面架构(Single-Page Architecture, SPA)的一部分, 因为 SPA 似乎是构建 Web 应用最有利和最受欢迎的方案。我会在 1.5.2 节中介绍如何将 React 集成到 SPA 中。

在某些极端情况下, 甚至可以在服务器端使用 React 作为模板引擎。例如, 有一个名为 express-react-views 的库(<https://github.com/reactjs/express-react-views>), 它在服务器端从 React 组件中渲染视图。这种服务器端渲染是可行的, 因为 React 允许使用不同的渲染目标。

### 1.5.1 React 类库和渲染目标

在版本 0.14 和更高的版本中, React 团队将库拆分成两个包: React Core(npm 上的 react 包)和 ReactDOM(npm 上的 react-dom 包)。React 维护者通过这种方式清楚地表明: React 不仅是一个 Web 库, 而且是正在变成一个描绘 UI 的通用库(有时称之为同构的, 因为它可以在不同环境中使用)。

例如, 在版本 0.13 中, React 使用 `React.render()` 方法将元素挂载到网页的 DOM 节点上。在版本 0.14 及更高的版本中, 需要依赖 `react-dom` 包, 并调用 `ReactDOM.render()` 方法而不是 `React.render()`。

React 社区创建了许多包来支持各种渲染目标, 这些包的存在使得这种分离组件书写和组件渲染逻辑的方法成为可能。其中一些模块如下:

- `blessed`(<https://github.com/chjj/blessed>)终端控制台接口渲染器: <http://github.com/Yomguithereal/react-blessed>
- 用于 ART 库的渲染器(<https://github.com/sebmarkbage/art>): <https://github.com/reactjs/react-art>
- 用于<canvas>的渲染器: <https://github.com/Flipboard/react-canvas>
- 使用了 `three.js` 的 3D 库渲染器(<http://threejs.org>): <https://github.com/lzzimach/react-three>
- 虚拟现实和 360° 互动体验渲染器: <https://facebook.github.io/react-vr>

除了这些库的支持外, React Core 和 ReactDOM 的分离使得在 React 和 React Native 库(用于原生移动 iOS 和 Android 开发)之间共享代码变得更加容易。实际上, 使用 React 进行 Web 开发时, 需要至少包含 React Core 和 ReactDOM。

此外, React 和 npm 中还有 React 的其他实用程序库。在 React v15.5 版本之前, 其中



一些被作为 React 附加组件<sup>15</sup>的一部分。这些实用程序库允许对功能进行增强、使用不可变数据(<https://github.com/kolodny/immutability-helper>)和进行测试。

最后, React 几乎总是和 JSX 一起使用, 这是一种小巧的语言, 可以让开发人员更加高效地编写 React UI。可以使用 Babel 或类似的工具将 JSX 转换为常规的 JavaScript。

如你所见, 和 React 相关的功能被分成很多不同的模块和 npm 包。这是一件好事, 让你有更多的能力和选择, 而不是通过准备一个完整的、大一统的库来为满足需求提供唯一可能的方式。更多内容会在 1.5.3 节中介绍。

如果你是一名 Web 开发者, 可能会使用 SPA 架构。也许你已经有一个 SPA 架构的 Web 应用, 并且希望使用 React 重新改造它; 也许你准备从头开始一个新项目。接下来, 我们将放大 React 在 SPA 中的地位, 把它作为最流行的 Web 应用构建方法。

## 1.5.2 单页面应用和 React

SPA 架构的别名是富客户端, 因为作为客户端的浏览器拥有更多的逻辑, 比如渲染 HTML、校验、UI 变更等功能。图 1.2 是一幅包含用户、浏览器和服务器的典型 SPA 架构图。图 1.2 中描绘了一个用户正在发出一个请求, 以及诸如单击按钮、拖曳、鼠标悬停等输入行为。

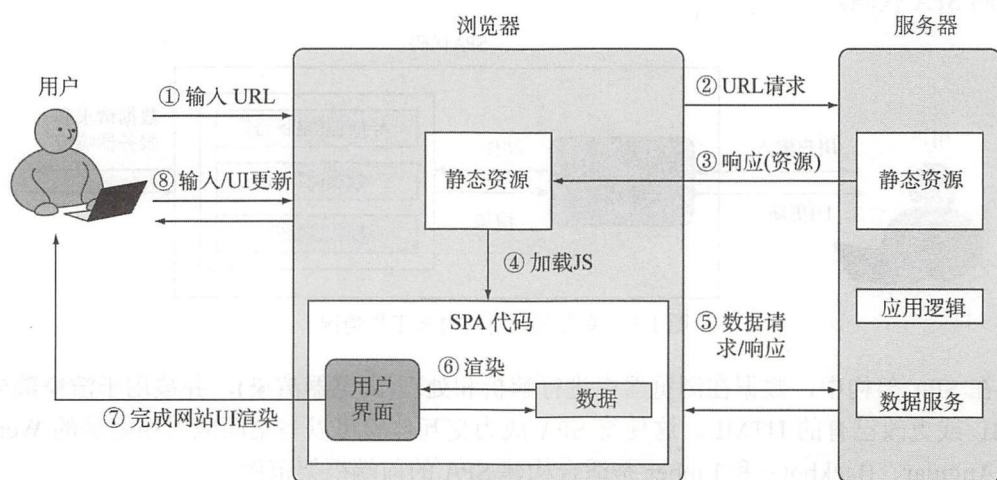


图 1.2 典型的 SPA 架构

- ① 用户在浏览器中输入一个 URL 以打开一个新的页面。
- ② 浏览器向服务器发送 URL 请求。
- ③ 服务器使用静态资源(如 HTML、CSS 和 JavaScript)进行响应。在大多数情况下, HTML 只是概要, 里面只有网页的骨架。通常会显示消息“加载中……”和/或转动的 GIF 图片。
- ④ 静态资源包括 SPA 中的 JavaScript 代码。加载时, 这些 JavaScript 代码对数据进行附加的请求(AJAX/XHR 请求)。

15 查看版本 15.5 的变更日志, 其中包含附加组件和 npm 库的清单: <https://facebook.github.io/react/blog/2017/04/07/react-v15.5.0.html>。也可以查看附加组件页面: <https://facebook.github.io/react/docs/addons.html>。

⑤ 数据以 JSON、XML 或任何其他格式返回。

⑥ 一旦 SPA 接收到数据, 就可以渲染缺失的 HTML 结构(图 1.2 中的用户界面模块)。换言之, UI 渲染发生在浏览器端, 并通过向 SPA 模板填充数据来完成<sup>16</sup>。

⑦ 一旦浏览器完成渲染, SPA 将替换消息“加载中……”用户便可以使用该页面了。

⑧ 用户看到一个漂亮的网页, 并可以与页面(图 1.2 中的输入)交互, 触发从 SPA 到服务器的新请求, 继续步骤②~⑥的循环过程。在这个阶段, 如果 SPA 实现了路由功能, 那么可能会触发路由的变更, 这意味着导航到一个新的 URL 会在浏览器中触发一次 SPA 渲染, 而不是从服务器端重新加载一个新的页面。

总而言之, 在 SPA 方式中, UI 的大部分渲染发生在浏览器端。只有数据往返于浏览器。相比而言, 使用富服务器端方式, 所有渲染都发生在服务器端(这里的渲染, 指的是从模板或 UI 代码生成 HTML, 而不是在浏览器中渲染 HTML(有时也称为绘制 DOM))。

请注意, 类 MVC 架构是实现 SPA 最流行的方式, 但并不是唯一的方式。React 不需要使用类 MVC 架构, 但为了简单起见, 我们假设你的 SPA 正在使用类 MVC 架构。可以在图 1.3 中看到其可能存在的不同部分。导航器或路由库作为 MVC 范例中的控制器, 指定要获取的数据和要使用的模板。导航器/控制器发起获取数据的请求, 然后使用数据对模板(视图)进行填充, 以 HTML 的形式渲染 UI。UI 会将操作(单击、鼠标悬停、敲击键盘等)发送回 SPA 代码。

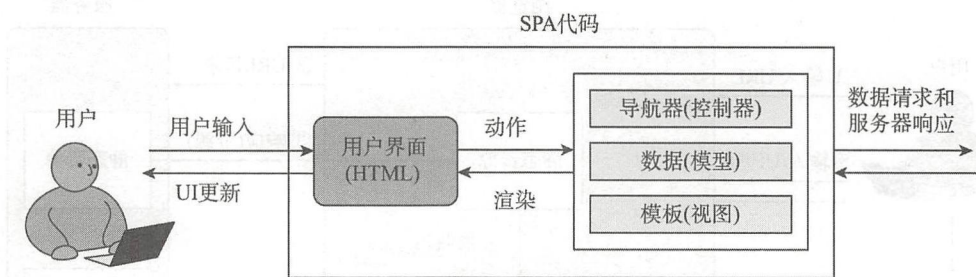


图 1.3 单页面应用的内部工作情况

在 SPA 架构中, 数据在浏览器中进行解析和处理(浏览器渲染), 并被用于渲染额外的 HTML 或更改已有的 HTML。这使得 SPA 成为交互体验可以与桌面应用相媲美的 Web 应用。Angular、Backbone 和 Ember 是适合构建 SPA 的前端框架范例。

注意: 不同的框架实现了不同的导航器、数据和模板, 因此图 1.3 并不适用于所有框架。相反, 图 1.3 只是阐述了典型 SPA 架构中最广泛的关注点分离。

React 在图 1.3 所示的 SPA 图表中所处的位置是模板方块。React 是一个视图层, 因此可以通过向其提供数据来使它渲染 HTML。当然, React 相比典型的模板引擎做的要多很多。React 和其他模板引擎(如 Underscore、Handlebars 和 Mustache)之间的区别在于开发、更新 UI 和管理状态的方式。我们将在第 4 章中更详细地讨论状态。现在, 将状态视为和 UI 相关的可更改数据即可。

16 “What does it mean to hydrate an object?”, Stack Overflow, <http://mng.bz/uP25>



### 1.5.3 React 技术栈

React 并不是一个完整的前端 JavaScript 框架。它非常简约，并不强制使用特定方式实现诸如数据模型、样式和路由等功能。因此，开发人员需要将 React 与路由和/或模型库搭配使用。

例如，已经使用 Backbone 和 Underscore 模板引擎的项目可以切换到 Underscore for React，并保留现有的数据模型和 Backbone 路由(Underscore 还包含了实用程序，而不仅仅包含模板方法，可以将这些 Underscore 实用程序与 React 一起用作声明式风格的解决方案)。其他时候，开发人员选择使用 React 技术栈，这个技术栈由专门为 React 创建的数据和路由库组成。

- 数据模型类库和后端：RefluxJS(<https://github.com/reflux/refluxjs>)、Redux(<http://redux.js.org/>)、Meteor(<https://www.meteor.com>)和 Flux(<https://github.com/facebook/flux>)
- 路由库：React Router(<https://github.com/ReactTraining/react-router>)
- 使用 Twitter Bootstrap 库的 React 组件集合：React-Bootstrap(<https://react-bootstrap.github.io>)

React 组件库的生态系统每天都在增长。React 描述可组合组件的能力非常有助于代码重用。有许多组件被打包成 npm 模块，只是为了证明拆分成小的可组合组件对于代码重用有好处，这里有一些受欢迎的 React 组件：

- 日期选择组件：<https://github.com/Hacker0x01/react-datepicker>
- 一套处理表单呈现和校验的工具：<https://github.com/prometheusresearch/react-forms>
- WAI-ARIA 兼容的自动完成组件：<https://github.com/reactjs/react-autocomplete>

接下来，我们聊聊 JSX，JSX 可能是最常见的不愿使用 React 的理由。如果熟悉 Angular，那么可能已经在模板代码中编写了大量的 JavaScript 代码。这是因为在现代 Web 开发中，纯 HTML 过于静态并且难以使用。我的建议是：对 React 的优点存疑，给 JSX 一次公平的展现机会。

JSX 使用和 XML/HTML 一样的语法，通过<>在 JavaScript 中编写 React 对象。React 和 JSX 配合得很好，因为开发人员可以更好地实现和阅读代码。JSX 可以看成编译成原生 JavaScript 的迷你语言，因此，JSX 并不在浏览器中运行，而是作为待编译的源代码。下面是用 JSX 编写的一段紧凑的代码：

```
if (user.session)
  return <a href="/logout">Logout</a>
else
  return <a href="/login">Login</a>
```

即便在浏览器中加载 JSX 文件，并通过运行时转换库在运行时将 JSX 编译为原生 JavaScript，也仍然不是运行 JSX，而是运行 JavaScript。在这一点上，JSX 类似于 CoffeeScript。将这些语言编译为原生 JavaScript 以获得相比 JavaScript 提供的更好的语法和功能。

对某些人而言，将 XML 插入到 JavaScript 代码中看起来非常怪异。我花了一段时间进行调整，因为我期待着一场雪崩式的语法错误消息。另外，JSX 是可选的。鉴于这两个原因，在第 3 章之前不会介绍 JSX；不过相信我，一旦掌握，功能会非常强大。



迄今为止，你已经了解了 React 是什么，它的技术栈以及它在更高层次 SPA 架构中的位置。现在是时候卷起袖子，开始编写你的第一个 React 项目了。

## 1.6 第一个 React 项目：Hello World

让我们开始探讨你的第一个 React 项目——学习编程语言的典型示例——HelloWorld 程序。这里不会使用 JSX，有的只是简单的 JavaScript。这个项目会在网页上打印内容为“Hello world!!!”的<h1>标题。图 1.4 展示了完成后的效果。



图 1.4 Hello World

### 首先学习不包含 JSX 的 React

尽管大多数 React 开发者都使用 JSX，但浏览器只能运行标准的 JavaScript。这也是理解纯 JavaScript 中的 React 代码为什么有益的原因。之所以从简单的 JS 开始，另一个原因是为了展示 JSX 是可选的，尽管它是 React 事实上的标准语言。最后一个原因是：JSX 的预处理需要一些工具。

我想尽快让你开始使用 React，而不用在本章中花费太多时间进行设置。在第 3 章中会执行 JSX 所有必要的设置。

该项目的文件夹结构很简单，它由 js 文件夹中的两个 JavaScript 文件和一个 HTML 文件 index.html 组成：

```
/hello-world
/js
  react.js
  react-dom.js
index.html
```

js 文件夹中的两个文件是 React v15.5.4<sup>17</sup>版本的：react-dom.js(Web 浏览器的 DOM 渲染器)和 react.js(React Core 包)。首先，需要下载上述 React Core 和 ReactDOM 库。方法有很多种，建议使用本书源代码中提供的文件，请访问 [www.manning.com/books/react-quickly](http://www.manning.com/books/react-quickly) 和 <https://github.com/azat-co/react-quickly/tree/master/ch01/hello-world>。这是最可靠且

<sup>17</sup> v15.5.4 是本书写作时的最新版本。通常，像 14、15 和 16 这样的主要版本间包含显著差异，而像 15.5.3 和 15.5.4 这样的小版本间变化和冲突较少。本书的代码在版本 v15.5.4 下进行了测试。这些代码可与未来的版本一起运行，但不能保证它们能正常运行。因为没有人知道未来的版本会是什么样的，甚至核心贡献者也不知道。

简单的方法，因为不需要依赖任何其他服务和工具。可以在附录 A 中找到下载 React 的更多方式。

**警告：**在版本 0.14 之前，这两个库是捆绑在一起的。例如，对于版本 0.13.3，只需要引入 `react.js` 即可。除非另有说明，本书使用版本 15.5.4(本书撰写时的最新版本)的 React 和 ReactDOM。对于本书第 I 部分的大多数项目，将需要两个文件：`react.js` 和 `react-dom.js`。在第 8 章中，将需要 `prop-types`([www.npmjs.com/package/prop-types](http://www.npmjs.com/package/prop-types))包，在版本 15.5.4 以前该包是 React 的一部分，但现在是一个单独的模块。

将 React 文件放在 `js` 文件夹中之后，在 Hello World 项目文件夹中创建 `index.html` 文件。这个 HTML 文件将是 Hello World 应用的入口(意味着需要在浏览器中打开)。

`index.html` 的代码很简单，首先在 `<head>` 标签中包含 React 相关的库。在 `<body>` 元素中，创建一个 `id` 属性为 `content` 的 `<div>` 容器，以及一个 `<script>` 标签(该应用的代码会写在这里)，如下所示：

代码清单 1.1 加载 React 库和代码(index.html)

```
<!DOCTYPE html>
<html>
  <head>
    <script src="js/react.js"></script>
    <script src="js/react-dom.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/javascript">
      ...
    </script>
  </body>
</html>
```

导入 React 库

导入 ReactDOM 库

定义一个空的 `<div>` 元素以便挂载 React UI

开始编写 Hello World 视图的 React 代码

为什么不直接在 `<body>` 元素中渲染 React 元素呢？因为这样做可能会导致与操作 `document body` 的其他库和浏览器扩展程序发生冲突。实际上，如果尝试把 React 元素挂载到 `body` 上，会得到如下警告：

```
Rendering components directly into document.body is discouraged...
```

这是 React 的另一个优点：提供很好的警告和错误消息！

**注意：**React 的警告和错误信息不包含在生产构建版本中，目的是减少干扰、增加安全性并尽可能减少发布大小。生产版本是 React Core 库的最小化版本，例如 `react.min.js`。有警告和错误消息的开发版本是未经最小化的版本，例如 `react.js`。

通过在 HTML 中引入如下两个库，可以访问 React 和 ReactDOM 这两个全局对象：`window.React` 和 `window.ReactDOM`。我们还需要两个方法：一个用于创建 React 元素(React)，另一个用于在 `<div>` 容器(ReactDOM)中渲染 React 元素，如代码清单 1.2 所示。

要创建 React 元素，需要做的就是调用 `React.createElement(elementName, data, child)`，三个参数的含义如下：



- **elementName**: HTML 标签字符串(例如'h1')或自定义的组件类对象(例如 2.2 节中的 HelloWorld)。
- **data**: 特性和属性数据(稍后会介绍), 例如 null 或 {name: 'Azat'}。
- **child**: 子元素或内部 HTML/文本内容, 例如 “Hello world!”。

#### 代码清单 1.2 创建和渲染 h1 元素(index.html)

```
var h1 = React.createElement('h1', null, 'Hello world!')
ReactDOM.render(
  h1,
  document.getElementById('content')
)
```

创建和保存一个存储在变量 h1 中的 React 元素

在 ID 为 content 的真实 DOM 元素中渲染 h1 元素

上面代码清单获取 h1 类型的 React 元素, 并将该对象引用存储到 h1 变量中。h1 变量不是实际的 DOM 节点, 而是 React 的 h1 组件(元素)的实例。可以以任意方式命名它, 例如 helloWordHeading。换言之, React 提供对 DOM 的抽象。

注意: h1 变量名是随意命名的。可以将其命名为任何想要的内容(例如 bananza), 只需要在 ReactDOM.render() 方法中使用相同的变量。

一旦元素被创建并存储在 h1 变量中, 就可以使用代码清单 1.2 所示的 ReactDOM.render() 方法, 将其渲染到 id 为 content 的 DOM 节点上。如果愿意, 也可以把 h1 变量的表达式移到 render 调用中, 除了不使用额外的变量之外, 结果是相同的:

```
ReactDOM.render(
  React.createElement('h1', null, 'Hello world!'),
  document.getElementById('content')
)
```

现在, 在你最喜欢的浏览器中打开由静态 HTTP 服务器发布的 index.html 文件。建议使用最新版本的 Chrome、Safari 或 Firefox 浏览器。你应该会在网页上看到 “Hello world!”, 如图 1.5 所示。

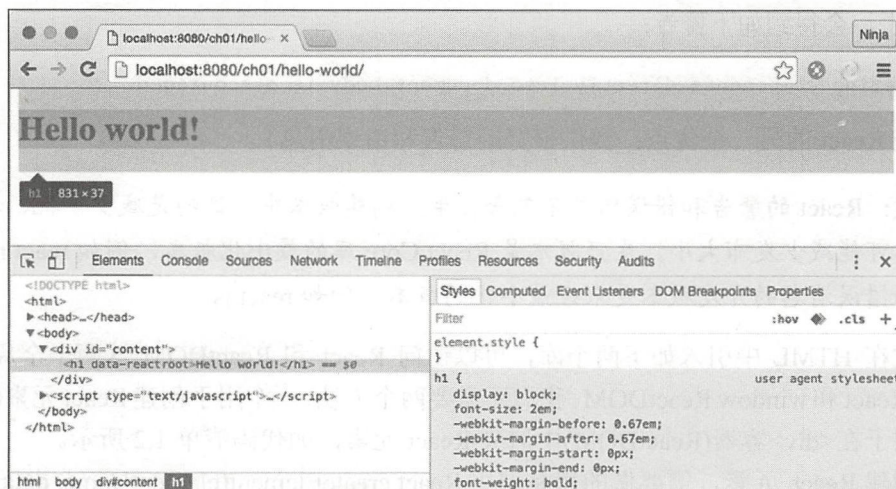


图 1.5 检查由 React 渲染的 Hello World 应用



图 1.5 展示了 Chrome 开发者工具的 Elements 选项卡并选择了 `<h1>` 元素。可以观察 `data-reactroot` 属性，它表示该元素由 ReactDOM 渲染。

需要注意的是：可以将 React 代码(代码清单 1.2)抽象为一个独立的文件，而不是创建一个元素，并在同一个 `index.html` 文件(代码清单 1.1)中使用 `ReactDOM.render()` 渲染它们。例如，可以创建 `script.js` 并将 `h1` 元素和 `ReactDOM.render()` 调用复制并粘贴到该文件中。然后在 `index.html` 中，需要在 `id` 为 `content` 的 `<div>` 标签后面插入 `script.js`，如下所示：

```
<div id="content"></div>
<script src="script.js"></script>
```

### 本地开发 Web 服务器

最好通过本地 Web 服务器访问 `index.html` 文件，而不是直接在浏览器中打开。因为使用 Web 服务器，JavaScript 应用可以生成 AJAX/XHR 请求。可以通过查看地址栏中的 URL 来判断是通过 Web 服务器访问还是直接访问文件。如果地址以 `file` 开头，就是直接访问文件；如果以 `http` 开头，就是通过 Web 服务器访问。在后续的项目中需要用到这个特性。通常，本地 HTTP Web 服务器会监听对 `127.0.0.1` 或 `localhost` 的请求。

可以使用任意的开源 Web 服务器，例如 Apache、MAMP(我的最爱，因为它们是使用 Node.js 开发的) `node-static`(<https://github.com/cloud-head/node-static>)或 `http-server`([www.npmjs.com/package/http-server](http://www.npmjs.com/package/http-server))。为了安装 `node-static` 或 `http-server`，需要安装 Node.js 和 npm。如果还没有它们，可以在附录 A 中找到 Node 和 npm 的安装说明，或者访问 <http://nodejs.org>。

假设你的机器上安装有 Node.js 和 npm，在命令提示符下运行 `npm i -g node-static` 或 `npm i -g http-server`。然后，切换到项目的源代码目录，运行 `static` 或 `http-server`。这里是从 `react-quickly` 目录开始执行的，所以需要在浏览器的地址栏中追加 `Hello World` 的路径：`http://localhost:8080/ch01/hello-world/`(见前面的图 1.5)。

恭喜！你刚刚实现了第一个 React 项目！

## 1.7 测验

1. 声明式编程不允许存储的值发生改变，这就是“要什么”，而不是命令式编程的“怎么做”。这么理解正确还是错误？
2. 要把 React 组件渲染到 DOM 中，使用哪个方法？(注意，这是个棘手的问题) `ReactDOM.renderComponent`、`React.render`、`ReactDOM.append` 还是 `ReactDOM.render`？
3. 必须在服务器上使用 Node.js 才能在 SPA 中使用 React。这么理解正确还是错误？
4. 必须包含 `react-dom.js` 才能在网页上渲染 React 元素。这么理解正确还是错误？
5. React 解决的问题是在数据变更时更新视图。这么理解正确还是错误？

## 1.8 小结

- React 是声明式的，它只是一个视图或 UI 层。

- React 使用组件，并且通过 `ReactDOM.render()` 方法对其进行渲染。
- React 组件类通过 `class` 创建，只有 `render()` 方法是必要的。
- React 组件是可重用的，并且拥有不可变的属性，可以通过 `this.props.NAME` 来访问它们。
- 在 React 中使用纯 JavaScript 来编写和组合 UI。
- 进行 React 开发时不必非得使用 JSX(针对 React 对象的一种类 XML 语法)，JSX 是可选的。
- 总结 React 的定义：针对 Web 的 React 由 React Core 库和 ReactDOM 库组成。React Core 库旨在通过使用 JavaScript 和(可选的)JSX，以同构的方式构建和共享可组合的 UI 组件。另一方面，要在浏览器中使用 React，可以使用 ReactDOM 库，该库拥有用于 DOM 渲染和服务器端渲染的方法。

## 1.9 测验答案

1. 正确。声明式风格的特点是“这是我想要的”，而命令式风格的特点是“这是怎么做的”。
2. `ReactDOM.render`。
3. 错误。可以使用任何后端技术。
4. 正确。需要 ReactDOM 库。
5. 正确。这正是 React 要解决的主要问题。

# 第 2 章

## React 起步

本章内容：

- 内嵌元素
- 创建组件类
- 使用属性

本章将教你如何开始使用 React，并为学习后续章节打下基础。理解 React 中的概念(例如元素和组件)至关重要。简言之，元素是组件(也称为组件类)的实例。请继续阅读以便了解它们的使用案例以及为什么要使用它们？

### 2.1 内嵌元素

在上一章中，你学习了如何创建一个 React 元素，使用的方法是 `React.createElement()`。例如，可以创建如下所示的链接元素：

```
let linkReactElement = React.createElement('a',  
  {href: 'http://webapplog.com'},  
  'Webapplog.com'  
)
```

问题是大多数 UI 都具有多个元素(如菜单中的链接)。例如在图 2.1 中，界面中有按钮、视频缩略图和 YouTube 播放器。

以分层方式创建更复杂结构的解决方案是使用内嵌元素。在上一章中，通过创建 React 元素 `h1` 并使用 `ReactDOM.render()` 将其渲染到 DOM 中，实现了第一个 React 项目：

```
let h1 = React.createElement('h1', null, 'Hello world!')
```



```
ReactDOM.render(  
  h1,  
  document.getElementById('content')  
)
```

值得注意的是, `ReactDOM.render()` 只使用一个元素作为参数, 示例中为 `h1`(图 2.2 展示了该视图)。

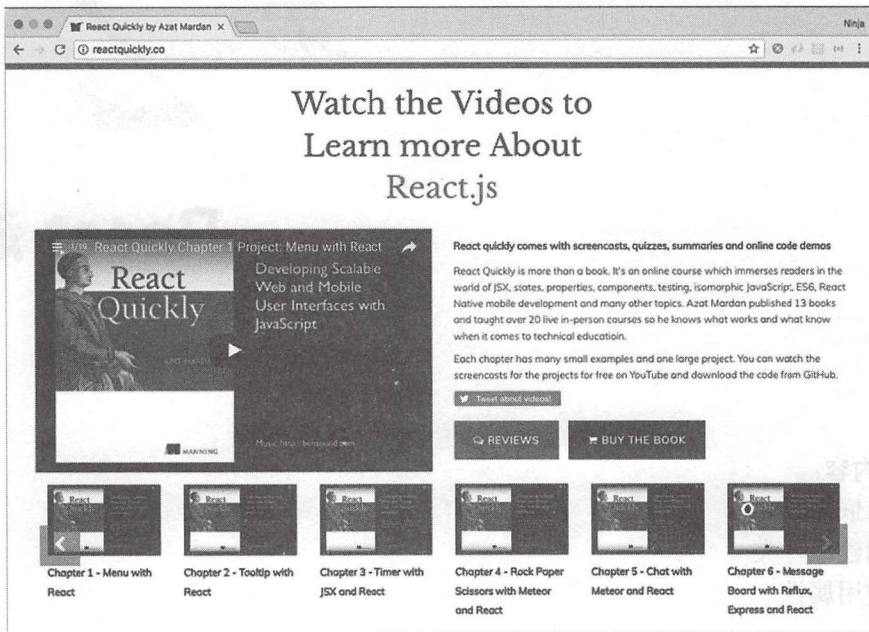


图 2.1 本书英文支持网站上有非常多的内嵌 UI 元素

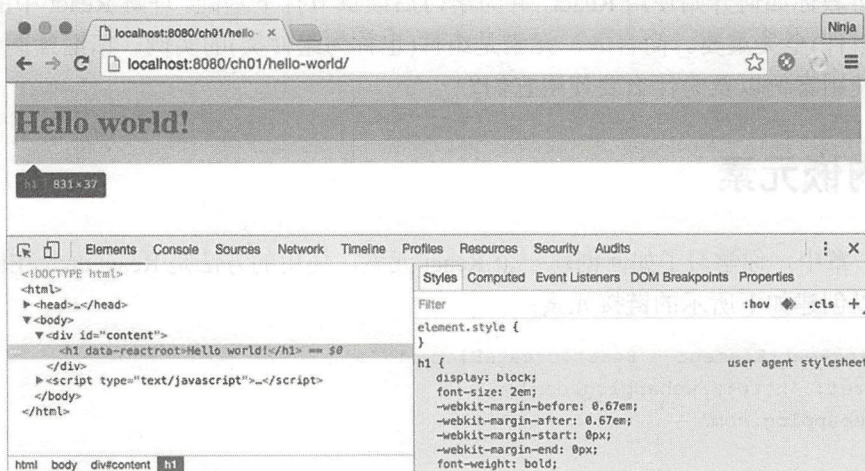


图 2.2 渲染一个单独的标题元素

本章开头提及, 当需要渲染两个同级别元素(例如两个 `h1` 元素)时, 会出现问题。这种情况下, 可将同级别元素包裹在视觉中性的元素中, 如图 2.3 所示。`<div>` 容器通常是很好的选择, `<span>` 也一样。



可以将数量不限的参数传递给 `createElement()`。第二个参数之后的所有参数都会成为子元素。这些子元素(本例中是 `h1`)是兄弟元素,也就是说,它们同级,例如图 2.4 中的 Chrome 开发者工具。

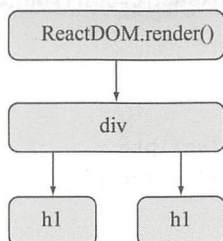


图 2.3 通过使用包裹的 `<div>` 容器渲染兄弟标题来构造一个 React `render` 返回对象



图 2.4 React 开发者工具展示了一个 `<div>` 标签, 里面包裹了两个内嵌的 `h1` 兄弟元素

### React 开发者工具

除了 Chrome 开发者工具中默认包含的 Element 选项卡之外, 还可以安装名为 React 开发者工具的扩展插件, 参见图 2.4 中的最后一个选项卡(React 选项卡)。Firefox 版本的 React 开发者工具也是可用的。它允许你仔细检查 React 渲染结果, 包括组件的层次结构、名称、属性、状态等。

GitHub 仓库地址为 <https://github.com/facebook/react-devtools>。也可以在 <http://mng.bz/V276> 上找到适用于 Chrome 浏览器的 React 开发者工具, 从 <http://mng.bz/59V9> 找到适用于 Firefox 版本的。

知道了这一点, 我们使用 `createElement()` 创建具有两个 `h1` 子元素(`ch02/hello-world-nested/index.html`)的 `<div>` 元素。

#### 代码清单 2.1 创建一个拥有两个 `<h1>` 子元素的 `<div>` 元素

```
let h1 = React.createElement('h1', null, 'Hello world!')
ReactDOM.render(
  React.createElement('div', null, h1, h1),
  document.getElementById('content')
)
```

如果第三个参数和后续参数不是文本, 那么它们指定了要创建元素的子元素

如果 `createElement()` 的第三个参数是一个字符串, 那么它指定了将被创建的元素内部的文本内容





HTML 代码可以沿用第 1 章的 Hello World 示例代码,只需要包含必要的 React 和 ReactDOM 库并包含 id 为 content 的 DOM 节点(ch02/hello-world-nested/index.html)

代码清单2.2 内嵌元素示例中不包含React代码的HTML文件

```
<!DOCTYPE html>
<html>
  <head>
    <script src="js/react.js"></script>
    <script src="js/react-dom.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/javascript">
      ...
    </script>
  </body>
</html>
```

迄今为止,只提供了字符串作为 `createElement()` 的第一个参数,但是第一个参数可以有两种类型的输入:

- 标准 HTML 标签名字符串,例如 'h1'、'div' 或 'p' (没有尖括号),名字小写。
- React 组件类对象,例如 HelloWorld,名字首字母大写。

第一种方式渲染标准 HTML 元素。React 查找标准的 HTML 元素列表,如果匹配到,就将其作为一种 React 元素类型来使用。例如,当传递 'p' 时,React 会匹配到,因为 'p' 是段落标签的名称。如果渲染这个 React 元素,就会在 DOM 中生成 `<p>`。

现在来看看第二种输入类型:创建和提供自定义组件类。

## 2.2 创建组件类

在使用 React 嵌套元素之后,会遇到下一个问题:很快就会拥有很多元素。你需要使用第 1 章中描述的基于组件的架构,它可以通过将功能分离为松散耦合的部件来重用代码。下面来看看组件类或者仅看看组件,它们经常被简单地调用(不要和 Web 组件混淆)。

把标准 HTML 标签想象成构建单元,可以使用它们来组成自己的 React 组件,可以使用这些组件来创建自定义元素(类的实例)。通过使用自定义元素,可以对便携式类(可组合组件和可重用组件)中的逻辑进行封装和抽象。这种抽象允许团队在大型复杂应用以及不同的项目中重用 UI。示例中包含了自动完成组件、工具箱、菜单等。

在 `createElement()` 方法中使用 HTML 标签创建 'Hello World' 元素非常直观: `createElement('h1', null, 'Hello World!')`。但是如果要像图 2.5 那样将 Hello World 分离到自己的类中,需要做什么呢?假设需要在 10 个不同的项目中重用 Hello World! (你可能不会多次使用它,但一个很好的自动填充组件肯定会被重用)。

有趣的是,你将使用 ES6 语法 `class CHILD extends PARENT`,通过继承 `React.Component` 类来创建一个 React 组件类。下面通过使用 `class HelloWorld extends React.Component` 来创





建一个自定义 HelloWorld 组件类。

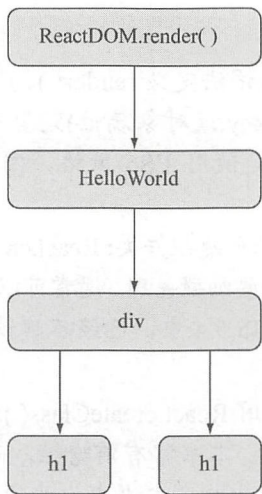


图 2.5 渲染从自定义组件类创建的一个<div>元素，替代直接进行渲染

代码清单 2.3(ch02/hello-world-class/js/script.js)展示了如何使用自定义 React 组件类 HelloWorld，将代码清单 2.1 中的嵌套 Hello World 示例重构为应用。这样做的好处是：通过自定义类，可以更好地重用这个 UI。Hello World 组件的强制方法 render() 返回与上一个示例相同的<div>元素。创建自定义 Hello World 类之后，可以将其作为对象(而不是字符串)传递给 ReactDOM.render()。

代码清单 2.3 创建和渲染一个 React 组件类

创建一个render()方法作为表达式(函数返回一个单一的元素)

把React元素绑定到真实的id属性值为content的DOM节点上

```
let h1 = React.createElement('h1', null, 'Hello world!')
class HelloWorld extends React.Component {
  render() {
    return React.createElement('div', null, h1, h1)
  }
}
ReactDOM.render(
  React.createElement(HelloWorld, null),
  document.getElementById('content')
)
```

定义一个首字母大写的React组件类

实现一条return语句，返回一个React元素，从而在React类调用render()方法时，可以返回一个带有两个h1元素的div元素

通过给第一个参数传递一个对象而不是字符串，使用Hello World类创建一个元素

依照惯例，包含 React 组件的变量的名称应该首字母大写，这在常规 JS 中并非必需的(例如使用小写变量名 helloWorld)。但在 JSX 中这么做非常必要，所以在此须遵从这个约定(在 JSX 中，React 通过大小写来区分普通 HTML 元素(例如<h1/>)和<HelloWorld />这样的自定义组件，但是在常规 JS 中，传递名为 HelloWorld 的变量和 'h1' 这样的字符串本身就不同。从现在开始就使用自定义组件的大小写约定是个不错的主意)。有关 JSX 的更多信息，



请参见第 3 章。

### ES6+/ES2015+和 React

组件类的示例使用 ES6 风格的语法定义 `render()`，其中省略了冒号和 `function` 关键字。效果和定义值为函数的属性(又称 `key` 或对象属性)完全一样，换言之，相当于输入 `render: function()`。我个人的喜好和建议是使用 ES6 风格，因为它更简短(输入越少，错误也就越少)。

历史上，React 有自己的方法用来创建组件类: `React.createClass()`。这个方法和使用 ES6 类继承 `React.Component` 的用法存在一些细微差别。通常可以使用类写法(推荐)或 `createClass()`，但不能同时使用。此外，在 React 15.5.4 中，已经不建议使用 `createClass()`(也就是不再支持了)。

虽然仍然可以看到一些团队使用 `React.createClass()` 方法，但 React 世界的趋势是朝着共同的标准迈进: 使用 ES6 类写法。本书富有前瞻性，而且使用最流行的工具和方法，所以专注于 ES6。可以在 GitHub 仓库中找到本书中一些项目的 ES5 示例，前缀为 `-es5`；它们是本书的早期版本。

截至 2016 年 8 月，大多数现代浏览器都已原生(无需额外的工具)支持这些 ES6 功能(和几乎所有其他功能)<sup>1</sup>，因此假设你已经很熟悉它。如果不是这样，或者需要温习 ES6+/ES2015+ 相关以及 React 相关的主要功能和更多信息，请参阅附录 E 或其他综合性书籍，如 Axel Rauschmayer 博士的 *Exploring ES6*(在线免费版: <http://exploringjs.com/es6>)。

与 `ReactDOM.render()` 类似，`createClass()` 中的 `render()` 方法只能返回单个元素。如果需要返回多个同级别的元素，请将它们包裹在 `<div>` 容器或其他非语义的标签元素(如 `<span>`)中。可以在浏览器中运行代码，结果如图 2.6 所示。

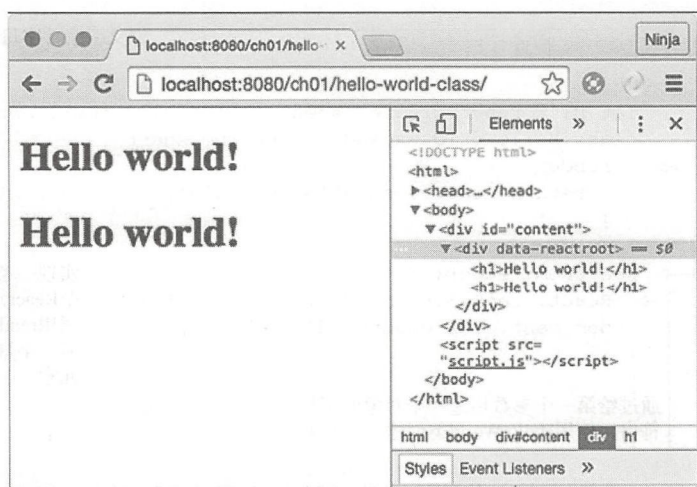


图 2.6 渲染通过自定义 HelloWorld 组件创建的元素

你可能会认为这种重构意义不大，但是如果需要打印更多的 HelloWorld 语句呢？可以通过重复使用 HelloWorld 组件并将其包装在 `<div>` 容器中来实现：

1 ECMAScript 6 兼容表格，<https://kangax.github.io/compat-table/es6>





```
...
ReactDOM.render(
  React.createElement(
    'div',
    null,
    React.createElement(HelloWorld),
    React.createElement(HelloWorld),
    React.createElement(HelloWorld)
  ),
  document.getElementById('content')
)
```

这就是组件可重用性的强大之处！可以使开发速度更快，产生的错误更少。组件还包括生命周期事件、状态、DOM 事件和其他特性，这些可以使组件更具交互性和独立性。这些会在后续章节中介绍。

现在，所有 `HelloWorld` 元素都是一样的，有办法可以定制它们吗？如果可以设置元素的属性并修改其内容和/或行为，又会怎样呢？一起来看看属性吧。

## 2.3 属性

属性是 React 使用声明式风格的基石。属性可以视为元素内的不可变值，如果在视图中使用它们，将允许元素具有不同的变化，例如通过传递新的属性值来更改链接的 URL：

```
React.createElement('a', {href: 'http://node.university'})
```

需要记住：属性在其组件内是不可变的。创建组件时，父组件为子组件分配属性。子组件不应该修改它的属性(子组件是嵌套在另一个元素中的元素；例如，`<h1/>` 是 `<HelloWorld />` 的子元素)。例如，可以为名为 `PROPERTY_NAME` 的属性传递值 `VALUE`，如下所示：

```
<TAG_NAME PROPERTY_NAME=VALUE/>
```

属性与 HTML 属性非常相似，这是设计目的之一，但还有另外一点：可以根据需要在代码中使用元素的属性。可以参照下面的方法使用属性：

- 为元素渲染标准的 HTML 属性：`href`、`title`、`style`、`class` 等。
- 在 React 组件类的 JavaScript 代码中通过 `this.props` 使用属性，例如 `this.props.PROPERTY_NAME`(用需要的任意名称替换 `PROPERTY_NAME`)。

在底层，React 会将属性名称(`PROPERTY_NAME`)与标准 HTML 属性列表相匹配。如果有匹配，属性会被渲染为 HTML 元素的属性(第一种情况)。属性的值也可以在组件类代码中通过 `this.props.PROPERTY_NAME` 进行访问。

如果与任何标准 HTML 属性名称不匹配(第二种情况)，就说明不是标准属性。它不会被渲染为 HTML 元素的属性，但是属性的值仍然可以在 `this.props` 对象中访问，例如 `this.props.PROPERTY_NAME`。可以在代码中使用或在 `render()` 方法中显式地渲染。这样，



可以将不同的数据传递给同一个组件类的不同实例。这允许对组件进行重用，因为可以通过编程的方式提供不同属性来修改元素的渲染方式。

### Object.freeze( )和 Object.isFrozen( )

在内部，React 使用 ES5 规范的 Object.freeze( )<sup>2</sup>来保证 this.props 对象不可变。要检查对象是否被冻结，可以使用 Object.isFrozen( )方法<sup>3</sup>。例如，可以查看下面这条语句是否返回 true:

```
class HelloWorld extends React.Component {
  render() {
    console.log(Object.isFrozen(this.props))
    return React.createElement('div', null, h1, h1)
  }
}
```

如果对更多细节感兴趣,建议阅读 React 变更日志<sup>4</sup>并在 React 的 GitHub 仓库<sup>5</sup>中进行搜索。

甚至可以更进一步地使用属性，根据属性的值来完全修改被渲染的元素。例如，如果 this.props.heading 为 true，就渲染“Hello”作为标题。如果为 false，就渲染“Hello”作为普通段落：

```
render() {
  if (this.props.heading) return <h1>Hello</h1>
  else return <p>Hello</p>
}
```

换言之，可以使用相同的组件，但属性不同，因而最终呈现的元素也可能不同。属性可以由 render( )渲染，在组件代码中使用，或者作为 HTML 属性使用。

为了演示组件的属性，我们用 props 稍稍修改一下 HelloWorld。目标是重用 HelloWorld 组件，使得该类的每个实例都呈现不同的文本和 HTML 属性。我们将使用三个属性来增强 HelloWorld 标题(<h1>标签)，如图 2.7 所示。

- id: 匹配标准 HTML 属性，并由 React 自动渲染。
- frameworkName: 与<h1>标签的任何属性都不匹配，但是会显式地展现在标题文本中。
- title: 匹配标准的 HTML title 属性，并由 React 自动渲染。

如果属性名和标准 HTML 属性相匹配，那么将被渲染为<h1>元素的属性，如图 2.7 所示。因此，id 和 title 这两个属性会被渲染为<h1>属性，而不会渲染为 frameworkName。甚至可能会收到关于未知 frameworkName 属性的警告信息(因为不在 HTML 规范中)。

2 Mozilla开发者网络, Object.freeze( ), <http://mng.bz/p6Nr>

3 Mozilla开发者网络, Object.isFrozen( ), <http://mng.bz/0P75>

4 GitHub, 2016-04-07-react-v15, <http://mng.bz/j6c3>

5 GitHub, “freeze” 搜索结果, <http://mng.bz/2l0Z>





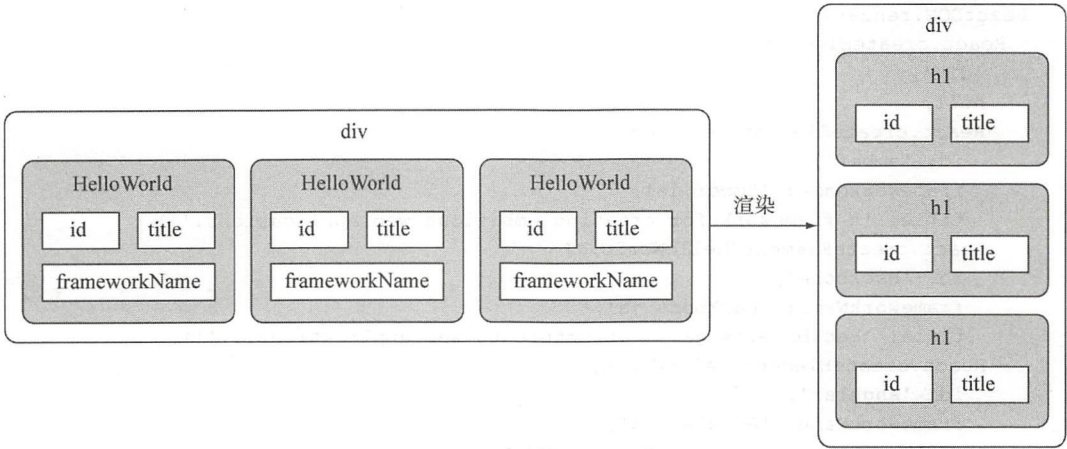


图 2.7 HelloWorld 组件类渲染标准的 HTML 属性，而不会渲染 frameworkName

让我们深入`<div>`元素的实现(见图 2.8)。显然，需要渲染 HelloWorld 类的三个子元素，但标题(`<h1 />`)的文本和属性必须不同。例如，传递 `id`、`frameworkName` 和 `title`，它们会成为 HelloWorld 类的一部分。

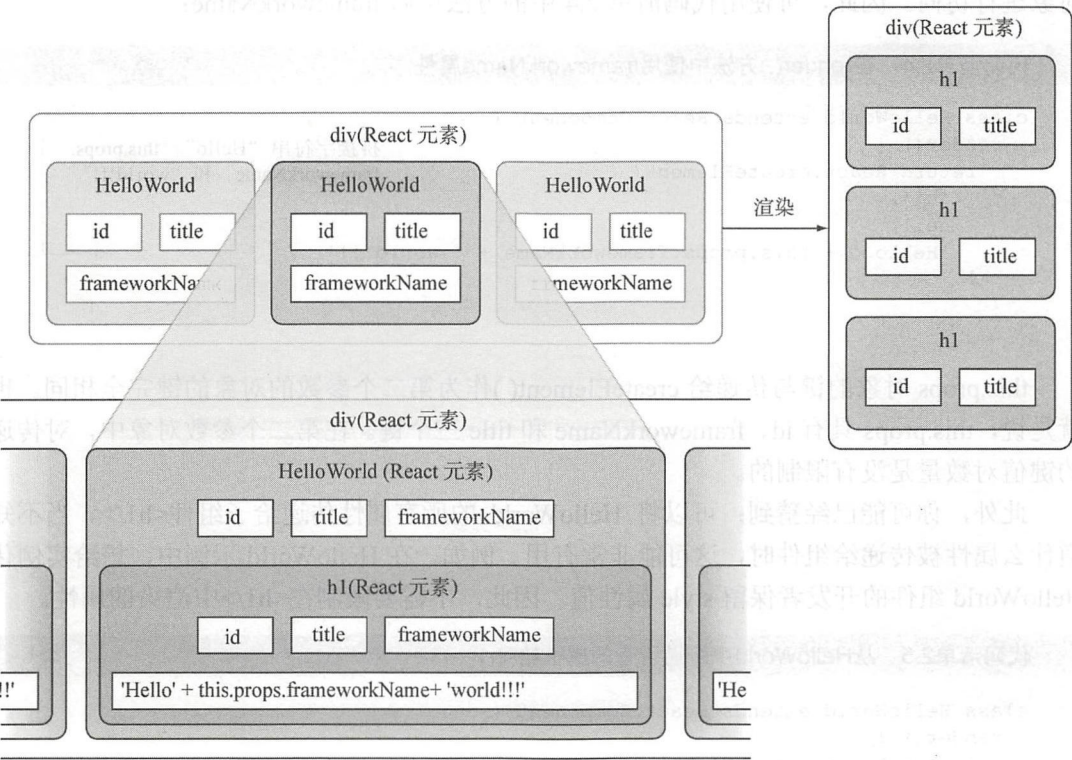


图 2.8 HelloWorld 类被使用了三次，以生成三个拥有不同 innerHTML 值的 h1 标签元素

在实现`<h1 />`之前，需要将属性传递给 HelloWorld。怎么做到这一点呢？当在`<div>`容器中创建 HelloWorld 元素时，将这些属性传递到 `createElement()` 的第二个参数对象中：



```
ReactDOM.render(
  React.createElement(
    'div',
    null,
    React.createElement(HelloWorld, {
      id: 'ember',
      frameworkName: 'Ember.js',
      title: 'A framework for creating ambitious web applications.'}),
    React.createElement(HelloWorld, {
      id: 'backbone',
      frameworkName: 'Backbone.js',
      title: 'Backbone.js gives structure to web applications...'}),
    React.createElement(HelloWorld, {
      id: 'angular',
      frameworkName: 'Angular.js',
      title: 'Superheroic JavaScript MVW Framework'})
  ),
  document.getElementById('content')
)
```

现在来看看 HelloWorld 组件的实现。React 的运作方式是：createElement()方法的第二个参数是一个对象，并且这个对象的属性可以在组件内部的 render()方法中通过 this.props 对象进行访问。因此，可使用代码清单 2.4 中的方法访问 frameworkName：

代码清单2.4 在render()方法中使用frameworkName属性

```
class HelloWorld extends React.Component {
  render() {
    return React.createElement(
      'h1',
      null,
      'Hello ' + this.props.frameworkName + ' world!!!'
    )
  }
}
```

拼接字符串“Hello”“this.props.frameworkName”和“world!!!”

this.props 对象的键与传递给 createElement()作为第二个参数的对象的键完全相同。也就是说，this.props 具有 id、frameworkName 和 title 三个键。在第二个参数对象中，对传递的键值对数量是没有限制的。

此外，你可能已经猜到：可以将 HelloWorld 的所有属性传递给子组件<h1/>。当不知道什么属性被传递给组件时，这可能非常有用。例如，在 HelloWorld 示例中，想给实例化 HelloWorld 组件的开发者保留 style 属性值。因此，不需要限制在<h1/>中渲染的属性。

代码清单2.5 从HelloWorld中传递所有的属性给<h1>

```
class HelloWorld extends React.Component {
  render() {
    return React.createElement(
      'h1',
      this.props,
      'Hello ' + this.props.frameworkName + ' world!!!'
    )
  }
}
```

传递所有属性给子组件标题元素





然后，将三个 HelloWorld 元素渲染到 id 为 content 的<div>中，如代码清单 2.6(ch02/hello-js-world/js/script.js)和图 2.9 所示：

代码清单2.6 在元素创建期间使用传递的属性

```
class HelloWorld extends React.Component {
  render() {
    return React.createElement(
      'h1',
      this.props,
      'Hello ' + this.props.frameworkName + ' world!!!'
    )
  }
}

ReactDOM.render(
  React.createElement(
    'div',
    null,
    React.createElement(HelloWorld, {
      id: 'ember', 3(CO5-3)
      frameworkName: 'Ember.js',
      title: 'A framework for creating ambitious web applications.'}),
    React.createElement(HelloWorld, {
      id: 'backbone',
      frameworkName: 'Backbone.js',
      title: 'Backbone.js gives structure to web applications...'),
    React.createElement(HelloWorld, {
      id: 'angular',
      frameworkName: 'Angular.js',
      title: 'Superheroic JavaScript MVW Framework'})
  ),
  document.getElementById('content')
)
```

把frameworkName属性作为文本输出到<h1>标签中

当createElement方法被调用时，HelloWorld组件的所有属性都会被传递到<h1>元素中

frameworkName不是<h1>的标准HTML属性，因此除非进行特殊处理，否则不会被渲染

id和title与<h1>的标准HTML属性相对应，并被渲染成这些属性

和之前一样，可以通过本地 HTTP Web 服务器运行代码。重用 HelloWorld 组件类的运行结果是三个不同的标题(见图 2.9)。

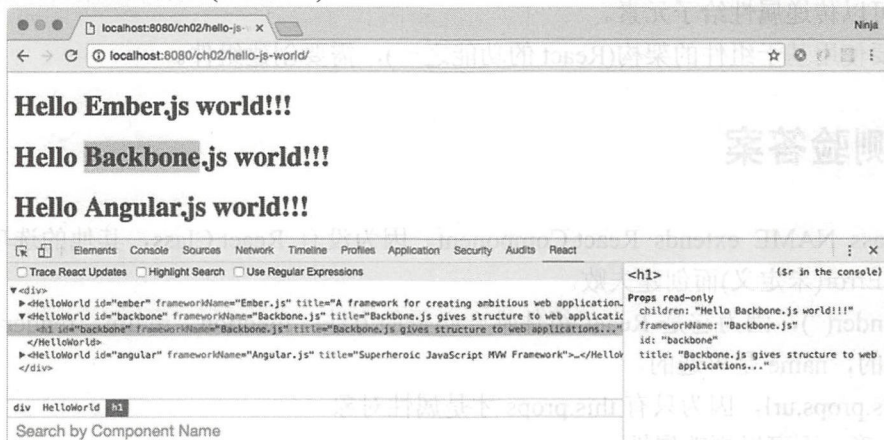


图 2.9 使用不同属性重用 HelloWorld 组件以渲染三个不同标题的运行结果

这里使用 this.props 为标题渲染不同的文本；使用属性渲染不同的 title 和 id。因此，通过有效地复用大部分代码，使你成为 React HelloWorld 组件类的主人！

我们已经涵盖了 Hello World 的几种排列。我知道，这仍然是一个无趣且老旧的 Hello World 示例。但是，不积跬步无以至千里，我们正在为未来介绍更高级的话题奠定坚实的基础。相信我，你可以通过组件类完成很多难办的事情。如果你(和许多 React 工程师一样)计划使用 JSX，那么知道 React 如何在常规 JavaScript 事件中运转非常重要。这是因为最终浏览器仍然会运行常规 JS，你需要了解从 JSX 到 JS 的转换结果。展望未来，我们将会使用 JSX，这在下一章中会介绍。

## 2.4 测验

1. 要创建一个 React 组件类，可以使用以下哪种方法？`createComponent()`、`createElement()`、`class NAME extends React.Component`、`class NAME extends React.Class`。
2. React 组件的唯一必需属性或方法是以下哪一项？`function`、`return`、`name`、`render`、`class`。
3. 要访问组件的 `url` 属性，可以使用以下哪一项？`this.properties.url`、`this.data.url`、`this.props.url`、`url`。
4. React 属性在当前组件的上下文中是不可变的，这么理解正确还是错误？
5. React 组件类允许开发人员创建可复用的 UI，这么理解正确还是错误？

## 2.5 小结

- 可以使用 `createElement()` 的第三个、第四个以及后续参数来嵌套 React 元素。
- 从自定义组件类创建元素。
- 使用属性修改结果元素。
- 可以传递属性给子元素。
- 要使用基于组件的架构(React 的功能之一)，需要创建组件。

## 2.6 测验答案

1. `class NAME extends React.Component`。因为没有 `React.Class`，其他的选项会因为 `ReferenceError`(未定义)而创建失败。
2. `render()`，因为它是 React 组件唯一必需的方法。`function`、`return`、`render` 和 `class` 都是非法的，`name` 是可选的。
3. `this.props.url`，因为只有 `this.props` 才是属性对象。
4. 正确。不可以更改属性。
5. 正确。开发人员使用新的组件来创建可重用 UI。



# 第 3 章

## JSX

本章内容：

- 了解 JSX 及其优点
- 使用 Babel 设置 JSX 编译器
- 了解 React 和 JSX 陷阱

欢迎来到 JSX 世界！在我看来，这是 React 最伟大的设计之一，也是在与我交谈 React 话题的几位开发者思想中最具争议的话题之一（如果不觉得惊讶，那是因为你还没有用 React 构建过任何大型的应用）。

迄今为止，已经介绍了如何创建元素和组件，以便于可以使用自定义元素并更好地组织 UI。我们使用 JavaScript 而不是使用 HTML 创建 React 元素。但是这么做会产生一个问题。我们来关注下面这段代码，看看能否发现点什么：

```
render() {  
  return React.createElement(  
    'div',  
    { style: this.styles },  
    React.createElement(  
      'p',  
      null,  
      React.createElement(  
        reactRouter.Link,  
        { to: this.props.returnTo },  
        'Back'  
      )  
    ),  
    this.props.children  
  );  
}
```

你是否能发现这里存在三个元素，它们是嵌套的，并且使用了 React Router 中的一个组件？与标准 HTML 相比，这段代码的可读性如何？是否经得起推敲？React 团队认为阅读(和键入)一堆 `React.createElement()` 语句并不是件有趣的事情，因此 JSX 被设计用来解决这个问题。

## 3.1 JSX 是什么？它有什么优点

JSX 是 JavaScript 的一种扩展，为函数调用和对象构造提供了语法糖，特别是 `React.createElement()`。JSX 看起来可能更像是模板引擎或 HTML，但它不是。JSX 生成 React 元素，同时允许你充分利用 JavaScript 的全部功能。

JSX 是编写 React 组件的极好方法，有以下优点：

- 改进的开发人员体验(Developer Experience, DX)：代码更易读，因为它们更加形象，感谢类 XML 语法，从而可以更好地表示嵌套的声明式结构。
- 更具生产力的团队成员：非正式开发人员(如设计师)可以更容易地修改代码，因为 JSX 看起来像 HTML，这正是他们所熟悉的。
- 更少的手腕磨损和语法错误：开发人员需要敲入的代码量变得更少，这意味着他们犯错的概率降低了，能够减少重复性伤害。

虽然 JSX 并不是 React 所必需的，但它非常适合 React，而且是 React 创建者们强烈推荐。官方的 JSX 简介页面<sup>1</sup>上说，“我们建议 React 和 JSX 一起使用”。为了演示 JSX 的表现力，我们创建 HelloWorld 和链接元素的代码：

```
<div>
  <HelloWorld/>
  <br/>
  <a href="http://webapplog.com">Great JS Resources</a>
</div>
```

这段代码和下面的 JavaScript 代码类似：

```
React.createElement(
  "div",
  null,
  React.createElement(HelloWorld, null),
  React.createElement("br", null),
  React.createElement(
    "a",
    { href: "http://webapplog.com" },
    "Great JS Resources"
  )
)
```

如果使用 Babel v6(转换 JSX 的一种工具，稍后会详细介绍)，JS 代码将变成：

```
"use strict";
```

---

1 <https://facebook.github.io/react/docs/introducing-jsx.html>



```

React.createElement(
  "div",
  null,
  " ",
  React.createElement(HelloWorld, null),
  " ",
  React.createElement("br", null),
  " ",
  React.createElement(
    "a",
    { href: "http://webapplog.com" },
    "Great JS Resources"
  ),
  " "
);

```

实质上，JSX 是一种类 XML 语法的小语言，但它改变了 UI 组件的编写方式。以前，开发人员以类似 MVC 的方式为控制器和视图编写 HTML 和 JS 代码，在各种文件之间跳转切换。这源于早期对关注点的分离，当 Web 应用只是为了使文本闪烁，并由静态 HTML、一点 CSS 和一小部分 JS 组成时，这种方法可以运作得很好。

如今时过境迁，我们需要构建具备强交互性的 UI，需要 JS 和 HTML 紧密耦合以实现各种功能。React 通过集成 UI 描述和 JS 逻辑来修复已然破损的关注点分离(Separation of Concern, SoC)原则。使用 JSX，代码看起来很像 HTML，并且更易于阅读和编写。如无其他原因，我会使用 React 和 JSX，只为这种新的 UI 编写方式。

JSX 由各种转换器(工具)编译成标准的 ECMAScript，如图 3.1 所示。你可能知道 JavaScript 就是 ECMAScript，但是 JSX 并非规范的一部分，它没有任何被规范定义的语义。

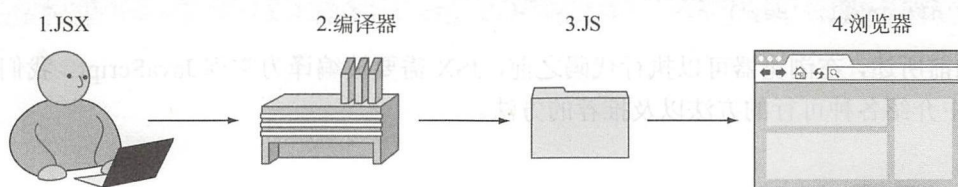


图 3.1 JSX 被编译成常规的 JavaScript

注意：根据 [https://en.wikipedia.org/wiki/Source-to-source\\_compiler](https://en.wikipedia.org/wiki/Source-to-source_compiler) 上的表述，“源代码到源代码的编译器、转换器或转译器”指代的都是一种编译器，以用一种编程语言编写的程序源代码作为输入，并生成另一种编程语言的等效源代码。

你或许会想，“为什么需要用 JSX？”，这个问题很好。考虑到初次看到 JSX 代码时会觉得有悖常理，许多开发人员会忽视这种惊人的技术也就不显得奇怪。例如，下面的 JSX 代码展示了 JavaScript 代码中有尖括号，刚开始这看起来非常诡异：

```
ReactDOM.render(<h1>Hello</h1>, document.getElementById('content'))
```

JSX 的令人惊叹之处就在于对 `React.createElement(NAME,...)` 的简写。可以使用 `<NAME />` 来替代一遍又一遍地编写该函数。正如刚才所说，输入越少，犯错就越少。使用 JSX，开

发体验(DX)与用户体验(UX)一样重要。

使用 JSX 的主要原因是:许多人发现使用尖括号(<>)比使用很多 `React.createElement()` 语句编写代码要更容易阅读。一旦习惯不把 `<NAME />` 看成 XML,而是把它作为 JavaScript 代码的别名,就会克服 JSX 语法带来的怪异感。当开发 React 组件及强大的 React 应用时,了解和使用 JSX 会带来很大不同。

#### 替代的快捷方式

为公平起见,和 JSX 对应,有一些替代方法可以避免输入 `React.createElement`,其中之一就是使用别名 `React.DOM.*`。例如,创建 `<h1 />` 元素:

```
React.createElement('h1', null, 'Hey')
```

下面的代码同样可以实现上述效果,而且需要的空间和时间更少:

```
React.DOM.h1(null, 'Hey')
```

可以访问 `React.DOM` 对象中的所有标准 HTML 元素,可以像检查任何其他对象一样检查它:

```
console.log(React.DOM)
```

也可以在 Chrome 开发者工具控制台中输入 `React.DOM` 并按回车键(请注意, `React.DOM` 和 `ReactDOM` 是两个完全不同的对象,不应该被混淆和互换使用)。

官方文档推荐的另一个替代方案是在 JSX 不可用的场景下(例如,当没有构建过程时),使用一个简短的变量。例如,可以创建变量 `E`,如下所示:

```
const E = React.createElement
E('h1', null, 'Hey')
```

如前所述,在浏览器可以执行代码之前,JSX 需要被编译为常规 JavaScript。我们将在 3.3 节中介绍各种可行的方法以及推荐的方法。

## 3.2 理解 JSX

接下来探讨如何使用 JSX。可以阅读本节并添加书签以便查找,或者如果希望在计算机上运行某些代码示例,你有以下选择:

- 在计算机上使用 Babel 设置一个 JSX 编译器,如 3.3 节所示。
- 使用在线 Babel REPL 服务(<https://babeljs.io/repl>),它可以在浏览器中把 JSX 编译成 JavaScript。

选择权在你手上,建议先阅读主要的 JSX 概念,然后在你的计算机上进行正确的 Babel 设置。



### 3.2.1 使用 JSX 创建元素

使用 JSX 创建 `ReactElement` 对象非常简单。例如，要实现如下 JavaScript(name 是一个字符串(h1)或组件类对象(HelloWorld))的功能：

```
React.createElement(  
  name,  
  {key1: value1, key2: value2, ...},  
  child1, child2, child3, ..., childN  
)
```

可使用如下 JSX：

```
<name key1=value1 key2=value2 ...>  
  <child1/>  
  <child2/>  
  <child3/>  
  ...  
  <childN/>  
</name>
```

在 JSX 代码中，属性及其值(例如，`key1=value1`)来自 `createElement()` 的第二个参数，本章的后面将重点介绍属性的使用。现在，我们来看一个没有属性的 JSX 元素示例。代码清单 3.1 是我们的老朋友：使用 JavaScript 编写的 Hello World(ch03/hello-world/index.html)。

代码清单3.1 使用JavaScript编写的Hello World

```
ReactDOM.render(  
  React.createElement('h1', null, 'Hello world!'),  
  document.getElementById('content')  
)
```

JSX 版本更加紧凑(ch03/hello-world-jsx/js/script.jsx)，如代码清单 3.2 所示。

代码清单3.2 使用JSX编写的Hello World

```
ReactDOM.render(  
  <h1>Hello world!</h1>,  
  document.getElementById('content')  
)
```

还可以将 JSX 语法创建的对象存储在变量中，因为 JSX 只是 `React.createElement()` 的语法糖而已，此例将对 `Element` 对象的引用存储在变量中：

```
let helloWorldReactElement = <h1>Hello world!</h1>  
ReactDOM.render(  
  helloWorldReactElement,  
  document.getElementById('content')  
)
```

### 3.2.2 在组件中使用 JSX

在前面的例子中使用了 JSX 标签<h1>，这也是一个标准的 HTML 标签名。使用组件时语法相同，唯一区别在于组件类名必须以大写字母开头，参见<HelloWorld />中的组件类。

代码清单 3.3 是使用 JSX 重写的、更高级的 Hello World 示例的迭代版本。在这里，将创建一个新的组件类并使用 JSX 创建元素。

代码清单3.3 使用JSX创建HelloWorld 类

```
class HelloWorld extends React.Component {
  render() {
    return (
      <div>
        <h1>1. Hello world!</h1>
        <h1>2. Hello world!</h1>
      </div>
    )
  }
}

ReactDOM.render(
  <HelloWorld />,
  document.getElementById('content')
)
```

对比下面的 JavaScript 代码，阅读代码清单 3.3 是否更加容易呢？

```
class HelloWorld extends React.Component {
  render() {
    return React.createElement('div',
      null,
      React.createElement('h1', null, '1. Hello world!'),
      React.createElement('h1', null, '2. Hello world!'))
  }
}

ReactDOM.render(
  React.createElement(HelloWorld, null),
  document.getElementById('content')
)
```

注意：如前所述，在 JavaScript 代码中看到尖括号可能对于有经验的 JavaScript 开发人员来说显得非常奇怪。当我第一次看到时，大脑几乎崩溃，因为许多年来我一直在训练自己发现 JS 的语法错误！尖括号是关于 JSX 的主要争议，也是我听到的最常见的反对意见之一，这就是为什么本书要尽早深入 JSX 的原因，你可以用它获取尽可能多的经验。

注意代码清单 3.3 所示 JSX 代码中 return 后面的圆括号。如果不打算再在同一行中输入任何内容，则必须加上它们。例如，如果在新行的开始输入根元素<div>，则必须在其外面包裹圆括号(( ))。否则，JavaScript 不会返回任何东西。这种格式如下所示：

```
render() {
```



```

    return (
      <div>
        </div>
    )
  }
}

```

也可以在 `return` 语句的同一行开始输入根元素，这样圆括号就不是必要的了。例如，下面的代码也是有效的：

```

render() {
  return <div>
    </div>
}

```

第二种方法的缺点是：起始 `<div>` 标签的可见性会降低，可能会被忽略<sup>2</sup>。选用哪种方法取决于你，本书将使用两种风格以便让你更为透彻地加以分析。

### 3.2.3 在 JSX 中输出变量

当组合组件时，你希望它们足够智能，可以根据某些变量自动更新视图。例如，显示当前日期时间的组件使用的应该是当前的日期和时间变量，而不是使用硬编码的值，这非常有用。

当使用纯 JavaScript 的 React 时，必须使用连接符(+)，或者使用 ES6+/ES2015+ 的由反引号和 `${varName}` 构成的字符串模板，其中 `varName` 是变量的名称。规范<sup>3</sup>中定义的这个功能的官方名称是模板字符串。例如，要在常规 JavaScript React 的 `DateTimeNow` 组件中使用文本中的属性，需要编写以下代码：

```

class DateTimeNow extends React.Component {
  render() {
    let dateTimeNow = new Date().toLocaleString()
    return React.createElement(
      'span',
      null,
      `Current date and time is ${dateTimeNow}.`
    )
  }
}

```

相反，在 JSX 中，可以使用花括号 `{}` 来动态输出变量，这可以极大地拟制代码量：

```

class DateTimeNow extends React.Component {
  render() {
    let dateTimeNow = new Date().toLocaleString()
    return <span>Current date and time is {dateTimeNow}</span>
  }
}

```

<sup>2</sup> 关于 JavaScript 中这一行为的更多信息，参见 James Nelson 于 2016 年 8 月 11 日在 <http://jamesknelson.com/javascript-return-parenthesis> 上发表的文章 “Why Use Parenthesis [sic] on JavaScript Return Statements?” 和 ECMAScript 5.1 规范中的 “Automated Semicolon Insertion” (<http://es5.github.io/#x7.9>)

<sup>3</sup> “模板字符串”，参见《ECMAScript 2015 语言规范》，2015 年 6 月，<http://mng.bz/i8Bw>

变量可以是 `props`，而不仅仅是本地定义的变量：

```
<span>Hello {this.props.userName}, your current date and time is  
  {new Date().toLocaleTimeString()}</span>
```

此外，可以在花括号 `{}` 中执行 JavaScript 表达式或任何 JS 代码。例如，格式化日期：

```
<p>Current time in your locale is  
  {new Date(Date.now()).toLocaleTimeString()}</p>
```

现在，可以使用存储在 JSX 变量中的动态数据重写 `HelloWorld` 类(ch03/hello-world-class.jsx)，参见代码清单 3.4。

代码清单 3.4 在 JSX 中输出变量

```
let helloWorldReactElement = <h1>Hello world!</h1>  
class HelloWorld extends React.Component {  
  render() {  
    return <div>  
      {helloWorldReactElement}  
      {helloWorldReactElement}  
    </div>  
  }  
}  
ReactDOM.render(  
  <HelloWorld/>,  
  document.getElementById('content')  
)
```

接下来我们将讨论如何在 JSX 中使用属性。

### 3.2.4 在 JSX 中使用属性

之前在介绍 JSX 时介绍过这个主题：元素的属性使用 HTML 属性语法来定义。也就是说，在 JSX 标签内使用 `key1=value1 key2=value2...` 的方式来定义 HTML 属性和 React 组件的属性，这和 HTML/XML 中的属性语法类似。

换言之，如果需要传递属性，可以像在正常 HTML 中那样在 JSX 中编写它们。另外，渲染标准 HTML 属性也是通过设置元素的属性来实现的(在 2.3 节中讨论过)。例如，下面的代码为锚点元素 `<a>` 设置了标准 HTML 属性 `href`：

```
ReactDOM.render((  
  <div>  
    <a href="http://reactquickly.co">Time for React?</a>  
    <DateTimeNow userName='Azat' />  
  </div>  
) ,  
  document.getElementById('content')  
)
```

渲染标准 HTML 属性 href

为 userName 属性设置值

对属性使用硬编码值非常不灵活。如果要重用链接组件，那么必须更改 `href` 以反映每次不同的地址。这被称为动态设置值而不是硬编码它们。因此，接下来我们将进一步介绍



可以使用动态生成的属性值的组件。这些值可以来自组件的属性(this.props)。之后，一切都会非常容易，需要做的所有工作就是在尖括号中使用花括号将属性的动态值传给元素。

例如，假设正在构建将用于链接到用户账户的组件。href 和 title 必须不同且不能硬编码。动态组件 ProfileLink 分别使用属性 url 和 label 为 href 和 title 渲染链接。在 ProfileLink 中，可以使用花括号 {} 将属性传给 <a>：

```
class ProfileLink extends React.Component {
  render() {
    return <a href={this.props.url}
      title={this.props.label}
      target="_blank">Profile
    </a>
  }
}
```

属性值来自哪里？它们是在创建 ProfileLink 时定义的，位于创建 ProfileLink 的组件中，也就是其父组件。例如，这就说明了在创建 ProfileLink 实例时 url 和 label 属性的值是如何传递的，这些值也就是渲染 <a> 标签的那些值：

```
<ProfileLink url='/users/azat' label='Profile for Azat' />
```

通过上一章，应该记住，渲染标准元素(<h>、<p>、<div>、<a>等)时，React 会从 HTML 规范中渲染所有属性，并忽略所有其他不属于规范的属性。这不是 JSX 的行为，而是 React 的行为。

但有时你会想要添加自定义数据作为属性。假设有一个列表项，有些信息对你的应用至关重要，但用户并不需要，通常的模式就是把这些信息放在 DOM 元素中作为属性。这个例子使用的是属性 react-is-awesome 和 id：

```
<li react-is-awesome="true" id="320">React is awesome!</li>
```

将数据存储 DOM 中的 HTML 自定义属性中通常被认为是一种反模式，因为并不希望 DOM 成为数据库或前端数据存储，从 DOM 获取数据比从虚拟/内存中要慢。

如果需要将数据存储为元素的属性，那么在 JSX 中，你需要使用 data-NAME 前缀。例如，要在 <li> 元素中渲染一个值为 this.reactIsAwesome 的自定义属性，可以这么写：

```
<li data-react-is-awesome={this.reactIsAwesome}>React is awesome!</li>
```

假设 this.reactIsAwesome 属性的值为 true，最终输出的 HTML 结果是：

```
<li data-react-is-awesome="true">React is awesome!</li>
```

但是，如果尝试将非标准的 HTML 属性传递给标准的 HTML 元素，那么该属性不会被渲染(如 2.3 节所述)。例如下面这段代码：

```
<li react-is-awesome={this.reactIsAwesome}>React is orange</li>
```

以及如下代码：

```
<li reactIsAwesome={this.reactIsAwesome}>React is orange</li>
```

输出的结果都是：

```
<li>React is orange</li>
```

显然，由于自定义元素(组件类)没有内置的渲染器并依赖标准的 HTML 元素或其他自定义元素，因此使用 `data-`前缀的问题对他们来说并不重要。它们会将所有属性作为 `this.props` 对象中的属性。

说到组件类，下面是使用常规 JavaScript 编写的 Hello World 示例的代码(见 2.3 节)：

```
class HelloWorld extends React.Component {
  render() {
    return React.createElement(
      'h1',
      this.props,
      'Hello ' + this.props.frameworkName + ' world!!!'
    )
  }
}
```

在 `HelloWorld` 组件中，不需要关注 `this.props` 中有哪些属性，而是直接全部传递给 `<h1>`。这在 JSX 中该如何实现呢？你肯定不想单独传递每一个属性，因为这样需要编写更多的代码。并且当属性变更时，这种紧耦合的代码也需要同步更新。试想一下，如果有两个或三个层级的组件需要手动传递每一个属性，该如何处理？这是一种反模式，不要像下面这样处理：

```
class HelloWorld extends React.Component {
  render() {
    return <h1 title={this.props.title} id={this.props.id}>
      Hello {this.props.frameworkName} world!!!
    </h1>
  }
}
```

当打算传递所有属性时，不要单个地传递，JSX 提供了一种扩展运算符的解决方案(...)，看起来像省略号，可以在下面的代码清单 3.5 中看到(ch03/jsx/hello-js-world-jsx)。

#### 代码清单3.5 使用属性

```
class HelloWorld extends React.Component {
  render() {
    return <h1 {...this.properties}>
      Hello {this.props.frameworkName} world!!!
    </h1>
  }
}

ReactDOM.render(
  <div>
    <HelloWorld
      id='ember'
```

```

    frameworkName='Ember.js'
    title='A framework for creating ambitious web applications.'/>,
<HelloWorld
  id='backbone'
  frameworkName= 'Backbone.js'
  title= 'Backbone.js gives structure to web applications...' />
<HelloWorld
  id= 'angular'
  frameworkName= 'Angular.js'
  title= 'Superheroic JavaScript MVW Framework' />
</div>,
document.getElementById('content')
)

```

借助`{...this.props}`，可以把每一个属性传递给子组件。其余的代码只是把 2.3 节中的例子转换成 JSX 版本。

### ES6+ / ES2015+中的省略号：rest 运算符、扩展运算符和解构

说到省略号，ES6+中有类似的运算符，叫作解构、扩展运算符和 rest 运算符。这是 JSX 使用省略号的原因之一！

如果曾经使用或编写具有变量或不限数量参数的 JavaScript 函数，那么应该知道 arguments 对象。这个对象包含传递给函数的所有参数。问题是这个 arguments 对象不是真正的数组。如果要显式地使用 `sort()` 和 `map()` 等函数，那么必须将其转换为数组。例如，下面的 request 函数使用 `call()` 方法转换 arguments 对象：

```

function request(url, options, callback) {
  var args = Array.prototype.slice.call(arguments, request.length)
  var url = args[0]
  var callback = args[2]
  // ...
}

```

在 ES6 中，有更好的方法把不限数量的参数作为数组来访问吗？答案是肯定的，以下就是 ES6 中 rest 参数 callbacks 的定义，callbacks 是一个真正的数组，而不是一个类似 arguments 对象的伪数组<sup>4</sup>：

```

function(url, options, ...callbacks) {
  var callback1 = callbacks[0]
  var callback2 = callbacks[1]
  // ...
}

```

rest 参数可以被解构，也就是说，它们可以被提取成独立的变量：

```

function(url, options, ...[error, success]) {
  if (!url) return error(new Error('oops'))
  // ...
}

```

4 在 rest 数组中，第一个参数没有名称：例如，callbacks 的下标是 0，而不是 2，就像 ES5 中的 arguments 对象。另外，在 rest 参数之后放置其他命名参数将导致语法错误



```

    success(data)
  }

```

那么扩展运算符呢？简言之，扩展运算符可以在以下位置展开参数或变量：

- 函数调用——例如 `push()` 方法：`arr1.push(...arr2)`
- 数组字面量——例如 `array2 = [...array1, x, y, z]`
- `new` 函数调用(构造函数)——例如 `var d = new Date(...dates)`

在 ES5 中，如果想要使用数组作为函数的参数，就必须使用 `apply()` 函数：

```

function request(url, options, callback) {
  // ...
}
var requestArgs = ['http://azat.co', {...}, function(){...}]
request.apply(null, requestArgs)

```

在 ES6 中，可以使用扩展运算符，和 `rest` 参数的语法一样，使用了省略号(`...`)：

```

function request(url, options, callback) {
  // ...
}
var requestArgs = ['http://azat.co', {...}, function(){...}]
request(...requestArgs)

```

扩展运算符的语法和 `rest` 参数的语法类似，但 `rest` 在函数定义/声明中使用，扩展运算符用于函数调用和字面量。它们可以避免输入额外的命令式代码，因此了解并使用它们是一项宝贵的技能。

### 3.2.5 创建 React 组件的方法

作为开发人员，可以为应用编写任何组件方法，因为 `React` 组件是类。例如，可以创建辅助方法 `getUrl()`：

```

class Content extends React.Component {
  getUrl() {
    return 'http://webapplog.com'
  }
  render() {
    ...
  }
}

```

`getUrl()` 方法并不复杂，但得到的启示是：可以创建自己的任意方法，而不仅仅是 `render()`。可以使用 `getUrl()` 方法抽象连接到 API 服务器的 URL。辅助方法可以具有可重用的逻辑，可以在组件的其他方法(包括 `render()`)的任何位置调用。

如果要从 `JSX` 中的自定义方法输出返回值，请使用 `{}`，就像使用变量一样(请参阅代码清单 3.6，`ch03/method/jsx/scrch03/meipt.jsx`)。这种情况下，在 `render()` 中调用辅助方法，方法的返回值将在视图中使用，记得使用 `()` 调用该方法。

## 代码清单3.6 调用一个组件方法以获取一个URL

```

class Content extends React.Component {
  getUrl() {
    return 'http://webapplog.com'
  }
  render() {
    return (
      <div>
        <p>Your REST API URL is:
          <a href={this.getUrl()}>      ← 在花括号中调用类的方法
            {this.getUrl()}
          </a>
        </p>
      </div>
    )
  }
  ...
}

```

再次强调，可以直接通过 JSX 和 `{ }` 调用组件的方法。例如，使用 `{this.getUrl( )}` 语法调用辅助方法 `getUrl`。当使用代码清单 3.6 中的方法时，将在段落 `<p>` 的链接中看到 `http://webapplog.com` 被作为返回值，返回如图 3.2 所示。

现在你应该已经了解组件的方法了。如果发现这部分内容太平淡，我表示歉意，因为这些方法作为 React 事件处理程序的基础非常重要。

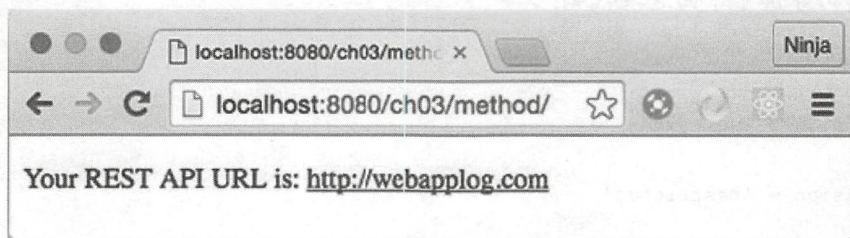


图 3.2 使用方法调用渲染 a 链接的结果

### 3.2.6 JSX 中的 if/else

和渲染动态变量类似，开发人员需要对组件进行组合以便组件可以根据 if/else 条件的结果更换视图。我们从一个简单的例子开始，它在组件类中渲染元素，渲染哪个元素取决于一个条件。例如，一些链接的文本和 URL 取决于 `user.session` 的值。下面展示了如何使用纯 JS 进行编码：

```

...
render() {
  if (user.session)
    return React.createElement('a', {href: '/logout'}, 'Logout')
  else
    return React.createElement('a', {href: '/login'}, 'Login')
}
...

```



可以使用一种相似的方式并使用 JSX 重写：

```
...
render() {
  if (this.props.user.session)
    return <a href="/logout">Logout</a>
  else
    return <a href="/login">Login</a>
}
...
```

假设还有其他元素，比如包裹在外的<div>。这种情况下，使用纯 JS，就必须创建一个变量，或者使用一个表达式或三元运算符(年轻的 JavaScript 开发人员称之为“猫王”运算符，参见 <http://mng.bz/92Zg>)，因为不能在<div>的 createElement()方法中使用 if 条件语句，而只能在运行时获取该值。

### 三元运算符

下面的三元条件运行逻辑是这样的：如果 userAuth 为 true，那么 msg 将被设置为 welcome，否则将被设置为 restricted。

```
let msg = (userAuth) ? 'welcome' : 'restricted'
```

上述语句等价于下面这段代码：

```
let session = ''
if (userAuth) {
  session = 'welcome'
} else {
  session = 'restricted'
}
```

某些情况下，三元运算符是 if/else 的简短版本。但是，如果尝试使用三元运算符作为表达式(需要返回一个值)，它们之间有很大的区别。下面这段代码是有效的 JS：

```
let msg = (userAuth) ? 'welcome' : 'restricted'
```

但是 if/else 并不能正常工作，因为这里不存在表达式，只有语句：

```
let msg = if (userAuth) {'welcome'} else {'restricted'} // Not valid
```

可以借助三元运算符的这个特性，在 JSX 运行时获取它的值。

为了演示三种不同的风格(变量、表达式、三元运算符)，请参阅下面的常规 JavaScript 代码，然后将它们转换为 JSX 版本：



```
// 方式1: Variable
render() {
  let link
  if (this.props.user.session)
    link = React.createElement('a', {href: '/logout'}, 'Logout')
  else
    link = React.createElement('a', {href: '/login'}, 'Login')
  return React.createElement('div', null, link)
}

// 方式2: Expression
render() {
  let link = (sessionFlag) => {
    if (sessionFlag)
      return React.createElement('a', {href: '/logout'}, 'Logout')
    else
      return React.createElement('a', {href: '/login'}, 'Login')
    }
  return React.createElement('div', null, link(this.props.user.session))
}

// 方式3: Ternary operator
render() {
  return React.createElement('div', null,
    (this.props.user.session) ? React.createElement('a', {href: '/logout'},
      'Logout') : React.createElement('a', {href: '/login'}, 'Login')
  )
}
```

使用link变量

← 创建一个表达式

使用三目运算符

这种方法不错，但有点笨拙。你同意这个观点吗？使用 JSX，符号 {} 可以打印变量并执行 JS 代码。让我们用它来实现更好的语法：

```
// 方式1: Variable
render() {
  let link
  if (this.props.user.session)
    link = <a href='/logout'>Logout</a>
  else
    link = <a href='/login'>Login</a>
  return <div>{link}</div>
}

// 方式2: Expression
render() {
  let link = (sessionFlag) => {
    if (sessionFlag)
      return <a href='/logout'>Logout</a>
    else
      return <a href='/login'>Login</a>
    }
  return <div>{link(this.props.user.session)}</div>
}

// 方式3: Ternary operator
render() {
  return <div>
    {(this.props.user.session) ? <a href='/logout'>Logout</a> :
      <a href='/login'>Login</a>}
  </div>
}
```

如果仔细查看表达式/函数风格的示例(方式 2: 在 `return` 之前, JSX 外部的一个函数), 可以想到一个替代方案。可以在 JSX 内使用立即调用的函数表达式(IIFE(Immediately Invoked Function Expression)), <http://mng.bz/387u>来定义相同的函数。这允许避免使用额外的变量(比如 `link`), 并在运行时执行 `if/else`:

```
render() {  
  return <div>{  
    (sessionFlag) => {  
      if (sessionFlag)  
        return <a href='/logout'>Logout</a>  
      else  
        return <a href='/login'>Login</a>  
    }(this.props.user.session)  
  }</div>  
}
```

定义 IIFE

使用参数调用 IIFE

此外, 还可以使用相同的原则来渲染文本和属性的值, 而不是渲染整个元素(在这些例子中是 `<a>`)。所需要做的就是花括号内使用这里所示的方式之一。例如, 可以增加 `url` 和文本, 而不是复用创建元素的代码。这是我个人最喜欢的方式, 因为可以使用单独的 `<a>`。

```
render() {  
  let sessionFlag = this.props.user.session  
  return <div>  
    <a href={ (sessionFlag) ? '/logout' : '/login' }>  
      { (sessionFlag) ? 'Logout' : 'Login' }  
    </a>  
  </div>  
}
```

创建一个本地变量来存储 `session` 的布尔值, 这样可以产生更少的代码和更好的性能

使用三目运算符来基于 `sessionFlag` 的值渲染不同的 URL

使用三目运算符来渲染不同的文本

如你所见, 与模板引擎不同的是, 在 JSX 中没有特殊的语法, 只是使用 JavaScript。通常情况下会使用三元运算符, 因为它是最紧凑的风格之一。总结一下: 当在 JSX 中使用 `if/else` 逻辑时, 可以使用以下选项:

- 在 JSX 外部(`return` 之前)定义变量并在 JSX 中使用 `{}` 打印。
- 在 JSX 外部(`return` 之前)定义表达式并在 JSX 中使用 `{}` 调用。
- 条件三元运算符。
- 在 JSX 中使用 IIFE。

当涉及条件和 JSX 时, 下面是我的经验法则: 在 JSX 之外使用 `if/else`(`return` 之前)生成一个变量, 在 JSX 中使用 `{}` 打印这个变量。或者省略这个变量, 直接在 JSX 中使用 `{}` 打印三元运算符或表达式的结果:

```
class MyReactComponent extends React.Component {  
  render() {  
    // Not JSX: Use a variable and if/else or ternary  
    return (  
      // JSX: Print result of ternary or expression with {}  
    )  
  }  
}
```

我们已经介绍了使用 React 和 JSX 构建交互式 UI 的重要条件，偶尔你也可能想为漂亮且智能的代码添加一些描述，以便其他人能快速理解。可以使用注释。

### 3.2.7 JSX 中的注释

JSX 中的注释与普通 JavaScript 注释类似，要在 JSX 中添加注释，可以使用 {} 包裹标准的 JavaScript 注释，如下所示：

```
let content = (  
  <div>  
    {/* Just like a JS comment */}  
  </div>  
)
```

也可以像下面这样使用注释：

```
let content = (  
  <div>  
    <Post  
      /* I  
      am  
      multi  
      line */  
      name={window.isLoggedIn ? window.name : ''} // We are inside of JSX  
    />  
  </div>  
)
```

现在，你已经品尝到 JSX 的好处了。本章的其余部分将致力于 JSX 工具和避免潜在的陷阱。没错：就是工具和陷阱。

在继续之前，你还必须了解，对于任何想要正常运行的 JSX 项目，需要对 JSX 进行编译。浏览器不能运行 JSX，它们只能运行 JavaScript，所以需要把 JSX 转换为普通的 JS 才行(参见图 3.1)。

## 3.3 使用 Babel 设置 JSX 转译器

如前所述，为了执行 JSX，需要将其转换为普通的 JavaScript 代码。这个过程被称为转译(源于编译和转换)，有非常多的工具可以用来完成这项工作。以下是推荐的方法：

- Babel 命令行接口(Command-Line Interface, CLI)工具：babel-cli 包提供了一个用于转译的命令。这种方法需要的设置较少，最易于启动。
- Node.js 或浏览器 JavaScript 脚本(API 方法)：脚本可以导入 babel-core 包并以编程的方式转译 JSX(babel.transform)。这种方法拥有更低阶的控制能力，并且移除了对构建工具及其插件的抽象以及依赖项。
- 构建工具：诸如 Grunt、Gulp 或 Webpack 这些工具可以使用 Babel 插件，这也是最受欢迎的方法。



所有这些方法都以不同的方式使用 Babel, Babel 主要是一个 ES6+/ES2015+编译器,但它也能把 JSX 转换成 JavaScript。实际上, React 小组已经停止开发自己的 JSX 转换器,并推荐使用 Babel。

可以使用 Babel 6 以外的方法吗?

虽然有各式各样的工具可以处理 JSX,但最常用并且由 React 团队于 2016 年 8 月在其官网上推荐的工具就是 Babel(之前称作 5to6)。历史上, React 团队维护着 react-tools 和 JSXTransformer(浏览器中的转译器),但是自从版本 0.13 开始, React 团队就开始推荐使用 Babel,并且停止继续更新 react-tools 和 JSXTransformer<sup>5</sup>。

对于浏览器中的运行时转译, Babel 5.x 有 browser.js,它是一个现成的发行版,可以把它放在浏览器中,就像 JSXTransformer 一样,它会把所有<script>标签(使用 type="text/babel")内的代码转译成 JS。最新的拥有 browser.js 的 Babel 版本是 5.8.34,可以直接从 CDN 引用(<https://cdnjs.com/libraries/babel-core/5.8.34>)。

Babel 6.x 不再包含默认的预设和配置(例如 JSX),并且移除了 browser.js。Babel 团队鼓励开发者创建它们自己的发行版本或者使用 Babel API。当然,还有一个 babel-standalone 库(<https://github.com/Daniel15/babel-standalone>),但是仍然要告诉它需要使用什么样的预设和配置。

Traceur(<https://github.com/google/traceur-compiler>)是另一个工具,可以用来代替 Babel。

最后, TypeScript([www.typescriptlang.org](http://www.typescriptlang.org))似乎通过 jsx-typescript 工具(<https://github.com/fdecampredon/jsx-typescript>)<sup>6</sup>支持 JSX 编译,但这是一种全新的工具链和语言(JavaScript 语言的超集)。

针对本书中的例子(使用 React v15),可以使用 JSXTransformer、Babel v5、babel-standalone、TypeScript 和 Traceur 工具。TypeScript 和 Traceur 应该相对安全,因为在本书撰写之时它们已经得到支持。但是如果使用 Babel 6 之外的其他工具,那么需要冒一下险。Manning 的技术审验人员和我都没有针对这些工具做过可用性测试。

在 React 中使用 Babel,可以获得额外的 ES6/ES2015 语法特性以简化开发,只需要添加额外的配置和 ES6 模块即可。ECMAScript 标准的第 6 次迭代有大量的改进,这些改进几乎在所有现代浏览器中都可以使用,但是在旧的浏览器中并不得到支持。另外,如果想使用 ES7、ES8 或 ES2017,一些浏览器可能还没有实现所有的特性。

在解决浏览器对 ES6 或 ES.NEXT 实现的滞后问题上, Babel 无疑是救世主。它支持下一代 JavaScript 语言(许多语言……是否从名字上得到了启示?)。本节介绍将在接下来几章中使用的推荐方法: Babel CLI,因为它需要的设置最少,并且不需要掌握 Babel API 的知识(不像 API 方式)。

使用 Babel CLI(<http://babeljs.io>),需要 Node v6.2.0、npm v3.8.9、babel-cli v6.9.0([www.npmjs.com/package/babel-cli](http://www.npmjs.com/package/babel-cli))和 babel-preset-react v6.5.0([www.npmjs.com/package/babel-preset-react](http://www.npmjs.com/package/babel-preset-react))。由于 Node.js 和 React 还在快速地更新迭代,因此其他版本并不能保证能正常运行本书的代码。

5 Paul O'Shannessy, "Deprecating JSTransform and react-tools" *React*, 2015/06/12, <http://mng.bz/8yGc>

6 [www.typescriptlang.org/docs/handbook/jsx.html](http://www.typescriptlang.org/docs/handbook/jsx.html)

如果需要安装 Node 和 npm, 最简单的方式就是从官网(<http://nodejs.org>)下载安装程序(一个包, 里面同时包括 Node 和 npm)。有关 Babel 安装的更多选项和详细安装说明, 请参阅附录 A。

如果已经安装了这些工具, 或者不确定是否安装, 请使用下面的 shell/终端/命令提示符 命令来检查 Node 和 npm 的版本:

```
node -v
npm -v
```

需要安装 Babel CLI 并且在本地配置 React 预设, 推荐做法是全局安装 Babel CLI(执行 npm 安装时使用 -g 选项), 因为当项目依赖 Babel CLI 的不同版本时, 可能会遇到冲突。以下是对附录 A 中简短版的说明:

(1) 创建一个新的文件夹, 例如 ch03/babel-jsx-test。

(2) 在这个新目录下创建 package.json 文件, 并输入一个空对象 {}, 或使用 npm init 生成该文件

(3) 在 package.json(本书使用的方案, 在下一节中详细阐述)或 .babelrc 文件(未在本书中使用)中定义 Babel 预设。

(4) 这一步是可选的, 完善 package.json 信息, 为之添加诸如项目名称、版权许可等信息。

(5) 本地安装 Babel CLI 和 React 预设, 使用 npm i babel-cli@6.9.0、babel-preset-react@26.5.0 --save-dev 命令安装这两个包, 并将依赖写入 package.json 文件的 devDependencies。

(6) 这一步是可选的, 使用一条精简的 Babel 命令创建一个 npm 脚本。

### Babel 的 ES6 预设

遗憾的是, 还必须支持诸如 IE9 的旧浏览器, 但是仍然希望使用 ES6+/ES2015+ 来书写代码, 因为这是未来的标准。可以添加 babel-preset-es2015 转译器([www.npmjs.com/package/babel-preset-es2015](http://www.npmjs.com/package/babel-preset-es2015))。它将把 ES6 代码转译成 ES5 代码。为此, 需要安装下面这个包:

```
npm i babel-preset-es2015--save-dev
```

然后将之添加到 React 的 presets 配置中:

```
{
  "presets": ["react", "es2015"]
}
```

如果不需要支持旧浏览器, 不建议使用这个 ES2015 转译器。有几个原因。首先, 将运行旧的 ES5 代码, 而这些代码并没有执行针对 ES6 的优化; 其次, 项目增加了额外的依赖项和复杂性; 最后, 如果大多数人继续在它们的浏览器中运行 ES5 代码, 为什么我们(浏览器开发团队和 JavaScript 开发者)还要使用 ES6 呢? 可以使用 TypeScript([www.typescriptlang.org](http://www.typescriptlang.org))、ClojureScript(<http://clojurescript.org>)或 CoffeeScript(<http://coffeescript.org>), 它们可以带来更多的好处。



要重复附录 A 中的步骤，需要至少包含如下预设信息的 `package.json` 文件：

```
{
  ...
  "babel": {
    "presets": ["react"]
  },
  ...
}
```

然后，执行如下命令(在新创建的项目文件夹中)来检查该版本是否工作：

```
$ ./node_modules/.bin/babel --version
```

安装之后，执行如下命令来将 JSX 代码 `js/script.jsx` 转译成常规 JavaScript 代码 `js/script.js`：

```
$ ./node_modules/.bin/babel js/script.jsx -o js/script.js
```

这个命令很长，因为使用了路径，可以将这个命令存储在 `package.json` 中，并使用简短的版本：`npm run build`。在编辑器中打开这个文件，并将下面这行代码添加到 `scripts` 中：

```
"build": "./node_modules/.bin/babel js/script.jsx -o js/script.js"
```

可以使用 `watch` 选项(`-w` 或 `--watch`)来实现自动转译：

```
$ ./node_modules/.bin/babel js/script.jsx -o js/script.js -w
```

当保存并更新 `script.jsx` 时，Babel 命令会监视 `script.jsx` 文件的变更，并把它编译成 `script.js`。当发生这种情况时，终端/命令提示符会打印如下信息：

```
change js/script.jsx
```

当积攒了很多 JSX 文件时，可以使用 `-d(--out-dir)` 和文件夹名称来将所有 JSX 源文件 (source) 编译成常规的 JS 文件 (build)：

```
$ ./node_modules/.bin/babel source --d build
```

通常，对于前端应用而言，加载一个文件比加载多个文件的性能更好，这是因为每个请求都会增加延迟。可以使用 `-o(--out-file)` 将源代码目录下的所有 JSX 文件编译成一个普通的 JS 文件：

```
$ ./node_modules/.bin/babel src -o script-compiled.js
```

可以使用 `babel` 替代 `./node_modules/.bin/babel`，这两种方式都是在本地执行，当然这取决于本机的路径配置。如果有一个全局安装的旧版本的 `babel-cli`，最好使用 `npm rm -g babel-cli` 将其移除。

在项目中，如果本地安装了 `babel-cli`，但是无法运行，可以考虑在 shell 概要文件中添加其中一条路径语句：`~/.bash_profile`、`~/.bashrc` 或 `~/.zsh`。如果使用的是 POSIX 操作系统 (UNIX、Linux、Mac OS 或类似操作系统)，则取决于你的 shell (bash、zsh 等)。

如下 shell 语句将添加一条路径，这样，如果在本地文件夹中存在 `./node_modules/.bin`



目录的话，就可以在不输入路径的情况下使用本地安装的 npm CLI 包：

```
if [ -d "$PWD/node_modules/.bin" ]; then
  PATH="$PWD/node_modules/.bin"
fi
```

shell 脚本会检查在当前命令行工具的工作路径下是否存在 `./node_modules/.bin` 文件夹，然后将这个文件夹添加到 `PATH` 环境变量中，以便允许诸如 Babel、Webpack 等 npm 命令行工具直接使用名字(babel、webpack 等)来运行。

也可以选择始终设置这条路径，而不仅仅是在有这个文件夹的情况下。如下 shell 语句将始终添加 `./node_modules/.bin` 路径到 `PATH` 环境变量中(也在配置文件中)：

```
export PATH="./node_modules/.bin:$PATH"
```

这个设置还允许在本地仅使用名称来运行任何 npm 命令行工具，而不是使用路径加名称的公式。

**提示：**要运行 Babel package.json 配置示例，请打开本书配套源代码中 `ch03` 文件夹中的项目。和后续各章中源代码的使用方法相同，在 `ch03` 文件夹的 `package.json` 文件中为每个需要编译的项目(子目录)创建 npm 构建脚本，除非该项目拥有自己的 `package.json` 文件。

当运行构建脚本时，例如 `npm run build-hello-world`，会将 `ch03/PROJECT_NAME/jsx` 目录下的 JSX 文件编译成常规 JavaScript 并输出到 `ch03/PROJECT_NAME/js` 目录中。因此，需要做的就是使用 `npm i`(这会创建 `ch03/node_modules` 目录)命令安装依赖，检查 `package.json` 文件中是否存在构建脚本，然后运行 `npm run build-PROJECT_NAME`。

迄今为止，你已经学习了将 JSX 编译成常规 JS 的最简单方法。但希望你能意识到，对于 React 和 JSX 还有一些棘手的部分。

## 3.4 React 和 JSX 陷阱

本节将介绍一些边界情况。当使用 JSX 时，有一些问题需要注意。

例如，在结束标记中 JSX 需要有一个结束斜杠(`/`)，如果没有子组件的话，可以在单个自闭合标签的后面跟一个斜杠。例如，下面的书写方式是正确的：

```
<a href="http://azat.co">Azat, the master of callbacks</a>
<button label="Save" className="btn" onClick={this.handleSave}/>
```

下面这些是不正确的，因为缺少斜杠：

```
<a href="http://azat.co">Azat<a>
<button label="Save" className="btn" onClick={this.handleSave}>
```

相反，HTML 的容错性更强，大多数浏览器会忽略缺失的斜杠，并在没有斜杠的情况下显示元素。自己试试看：`<button>Press me.`

当然，HTML 和 JSX 之间还有一些其他区别。

### 3.4.1 特殊字符

HTML 实体是一些显示特殊字符的编码, 例如版权符号、破折号、引号等。示例如下:

```
&copy;  
&mdash;  
&ldquo;
```

可以在`<span>`标签内或`<input>`标签的字符串属性中渲染这些字符, 例如下面的静态 JSX(标签文本使用这些编码定义, 并未使用变量或属性):

```
<span>&copy; &mdash; &ldquo;</span>  
<input value="&copy; &mdash; &ldquo;" />
```

但是, 如果想在`<span>`标签中动态输出 HTML 实体(从变量或属性中输出), 将会得到直接的文本输出(`&copy;&mdash;&ldquo;`)而不是特殊字符。因此, 下述代码无法正常工作:

```
// Anti-pattern. Will NOT work!  
var specialChars = '&copy; &mdash; &ldquo;'  
  
<span>{specialChars}</span>  
<input value={specialChars} />
```

React/JSX 会自动规避危险的 HTML, 这在确保安全性方面非常方便。要输出特殊字符, 可以使用下述方法之一:

- 把它们切分成一个字符串数组的多个元素; 例如: `<span>{[<span>&copy; &mdash; &ldquo;</span>]}</span>`。还可以设置 `key` 属性, 例如, `key="specialChars"`, 以规避缺少键的警告信息。
- 把特殊字符直接拷贝到源代码中(确保 UTF 在使用 UTF-8 字符集)。
- 使用 `\u` 加上 Unicode 编码来转义特殊字符(如果不记得, 可以在 [www.fileformat.info/info/unicode/char/search.htm](http://www.fileformat.info/info/unicode/char/search.htm) 上检索)。
- 使用 `String.fromCharCode(charCodeNumber)` 从字符 Unicode 编码转换而来。
- 使用内建方法 `__html` 设置危险的内部 HTML(<http://mng.bz/Tp1O>; 不推荐)。

为了论证最后一种方法(所有方法都失败时的保留方法), 请看下面的代码:

```
var specialChars = {__html: '&copy; &mdash; &ldquo;'  
<span dangerouslySetInnerHTML={specialChars} />
```

显然, React 团队很幽默, 他们以 `dangerouslySetInnerHTML` 来命名这个属性。我经常被 React 的命名逗乐。

### 3.4.2 data-属性

2.3 节介绍了非 JSX 方法中的属性, 下面让我们看看如何在 HTML 中再次创建自定义属性(这一次使用 JSX)。最主要的是, React 会愉快地忽略添加到组件的任何非标准 HTML 属性。无论使用的是 JSX 还是原生的 JavaScript, 都一样, 这就是 React 的行为。

但有时, 希望通过 DOM 节点传递额外的数据, 这是反模式, 因为 DOM 不应该被用

作数据库或本地存储。如果仍然希望创建自定义属性并渲染, 请使用 `data-`前缀。

例如, 下面是一个有效的自定义 `data-object-id` 属性, `React` 会将其渲染到视图中(HTML 会和 JSX 一样):

```
<li data-object-id="097F4E4F">...</li>
```

如果输入是以下 `React/JSX` 元素, 那么 `React` 不会渲染 `object-id`, 因为它不是标准的 HTML 属性(HTML 不像 JSX, 不存在 `object-id`):

```
<li object-id="097F4E4F">...</li>
```

### 3.4.3 style 属性

JSX 中的 `style` 属性和普通的 HTML 不同。在 JSX 中需要传递 JavaScript 对象而不是字符串, CSS 属性命名必须使用小驼峰风格。例如:

- `background-image` 变成 `backgroundImage`
- `font-size` 变成 `fontSize`
- `font-family` 变成 `fontFamily`

可以定义一个变量来存储这个 JavaScript 对象, 或者以内联方式在双花括号(`{{...}}`)中渲染它。这里需要使用双花括号, 因为有一对是 JSX 需要的, 另一对用于 JavaScript 对象字面量。

假设定义了一个包含 `fontSize` 属性的对象:

```
let smallFontSize = {fontSize: '10pt'}
```

在 JSX 代码中, 可以直接使用 `smallFontSize` 对象:

```
<input style={smallFontSize} />
```

也可以在 JSX 中直接给定一个对象以设置更大的字体(30 点)而不需要定义额外的变量:

```
<input style={{fontSize: '30pt'}} />
```

来看另一个直接传递样式的例子, 这次在 `<span>` 上设置了一条红色边框:

```
<span style={{borderColor: 'red',  
  borderWidth: 1,  
  borderStyle: 'solid'}}>Hey</span>
```

或者, 下面的边框值也是有效的:

```
<span style={{border: '1px red solid'}}>Hey</span>
```

使用 JavaScript 对象而不是字符串的主要原因是: 当视图发生变更时, 使用对象, `React` 可以渲染得更快。



### 3.4.4 class 和 for

React 和 JSX 接受任何标准的 HTML 属性,除了 `class` 和 `for`。这两个单词是 JavaScript/ECMAScript 保留字。React 使用 `className` 和 `htmlFor` 来代替它们。例如,如果有一个名为 `hidden` 的 class,可以在 `<div>` 中这样定义:

```
<div className="hidden">...</div>
```

如果需要为一个表单元素创建一个标签,使用 `htmlFor`:

```
<div>
  <input type="radio" name={this.props.name} id={this.props.id}>
</input>
  <label htmlFor={this.props.id}>
    {this.props.label}
  </label>
</div>
```

### 3.4.5 布尔类型的属性值

最后,但并非次要的是,某些属性(如 `disabled`、`required`、`checked`、`autofocus` 和 `readOnly`)只能在表单元素上使用。这里最重要的一点是:属性值必须设置在 JavaScript 表达式中(在 `{ }` 里),并且不能设置成字符串。

例如,使用 `{false}` 来启用 `input` 元素:

```
<input disabled={false} />
```

但是不要直接使用 `"false"` 值,因为这样会传递值 `true`(非空字符串在 JavaScript 中被认为是 `true`),并且渲染一个不可用的 `input` 元素:

```
<input disabled="false" />
```

#### 真假判断

在 JavaScript/Node 中,当真值被当成布尔值进行计算时,会被翻译为 `true`。例如,在一条 `if` 语句中,如果一个值非假,那它就是真值(这是官方定义)。一共有 6 个假值。

- `false`
- `0`
- `""`(空字符串)
- `null`
- 未定义
- `NaN`

可以看到,字符串 `"false"` 并非非空字符串,而是真值,并且会被翻译成 `true`。因此,你会在 HTML 中得到 `disabled=true`。

如果不设置这个值，React 会假设这个值为 `true`：

```
<input disabled />
```

后续各章将只会使用 JSX。但是，知道实际在浏览器中运行的转译后的常规 JavaScript 将是你技能集中的重要技巧。

## 3.5 测验

1. 要在 JSX 中输出一个 JavaScript 变量，你会使用下列哪种方式？`=`、`<%= %>`、`{}` 还是 `<?= ?>`？
2. 在 JSX 中不允许有 `class` 属性，这样理解正确还是错误？
3. 没有赋值的属性的默认值是 `false`，这样理解正确还是错误？
4. JSX 中的内联样式属性是 JavaScript 对象，并且不是类似字符串的属性，这样理解正确还是错误？
5. 如果需要在 JSX 中使用 `if/else` 逻辑，可以在 `{}` 中使用。例如，`class={if(!this.props.admin) return 'hide'}` 是合法的 JSX 代码，这样理解正确还是错误？

## 3.6 小结

- JSX 只是 React 的 `createElement()` 方法的语法糖。
- 应该使用 `className` 和 `htmlFor` 来替代标准的 `class` 和 `for` 属性。
- `style` 属性的值应该是 JavaScript 对象，而不是类似普通 HTML 的字符串。
- 三元运算符和 IIFE 是实现 `if/else` 语句的最佳方法。
- 输出变量、注释和 HTML 实体，并且编译 JSX 代码为原生 JavaScript 很容易。
- 有一些方法可以将 JSX 编译成常规 JavaScript，和使用诸如 Gulp 或 Webpack 这类构建工具以及使用 Babel API 编写 Node/JavaScript 脚本相比，使用 Babel CLI 所需的设置最少。

## 3.7 测验答案

1. 使用 `{}` 输出变量和表达式。
2. 正确。`class` 是保留字或特殊的 JavaScript 语法。因此，在 JSX 中应该使用 `className`。
3. 错误。建议使用 `attribute_name={false/true}` 来显式地设置布尔值。
4. 正确。出于性能考虑，样式是对象。
5. 错误。首先，`class` 不是合适的属性名。其次，`if` 语句是非法的，应该使用三元运算符。

# 第4章

## 与状态交互

### 本章内容:

- 了解 React 组件的状态
- 使用状态
- 状态和属性
- 有状态和无状态组件

如果本书只读一章的话，那无疑就是本章。没有状态，React 组件只是一堆静态模板而已。希望你和我一样兴奋，因为理解本章中的概念有助于构建更有趣的应用。假设正在构建一个自动填充的输入表单，如图 4.1 所示。输入内容时，希望向服务器发送请求以获取匹配的信息，以便显示在网页上。迄今为止，我们已经使用了属性，已经了解到通过更改属性可以获得不同的视图。但是在当前组件的上下文中，属性不可更改，因为它们是在创建组件时传递进来的。

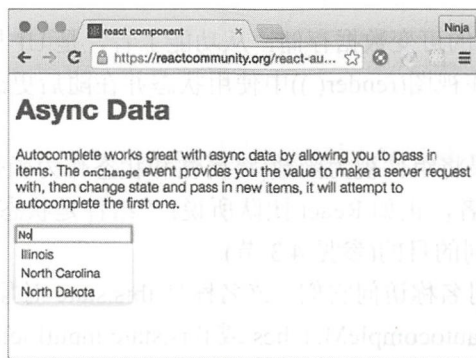


图 4.1 交互中的 react-autocomplete 组件



换言之，属性在当前组件内是不可变的，这意味着不能更改组件的属性，除非通过父组件传递新值来重新创建组件，如图 4.2 所示。但是从服务器收到的信息又必须存储在某处，然后在视图中显示新的匹配列表。如果属性无法更改，应该如何更新视图呢？

一种解决方案是在每次获得新的服务器响应时渲染具有新属性的元素。但是，这样就必须在组件外部编写逻辑——组件将不再是自包含组件。显然，如果无法更改属性，并且自动完成组件需要自包含，就不能使用属性。因此，问题是如何更新视图以响应事件而无须重新创建组件(`createElement()`或`JSX<NAME/>`)？这正是状态要解决的问题。

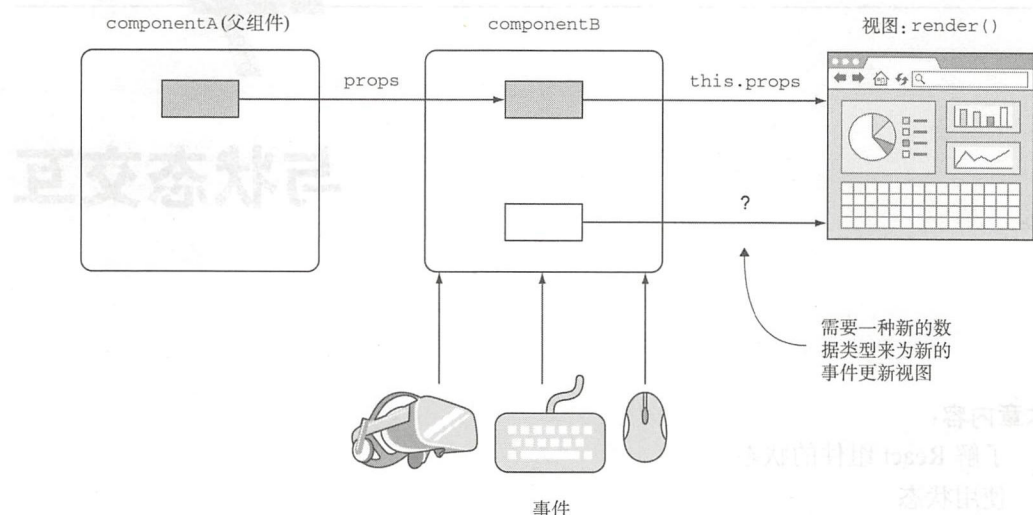


图 4.2 我们需要另一种在组件中可变的类型来控制视图更新

一旦服务器响应准备就绪，回调代码将相应地增加组件状态，这段代码必须自己编写。一旦状态被更新，React 将会非常智能地更新视图(仅更新需要更新的地方，也就是使用了状态数据的地方)。

使用 React 组件状态，可以构建有意义的、交互式的 React 应用。状态是一个核心概念，允许构建可以存储数据的 React 组件，并根据数据更改自动更新视图。

## 4.1 什么是 React 组件的状态

React 状态是组件内部的可变数据存储，从功能上看，是 UI 与逻辑的核心。可变意味着状态值可以改变。通过在视图(`render()`)中使用状态并在随后更改其值，可以影响视图的展现。

这里有如下隐喻：如果将组件视为函数(将属性和状态作为函数的输入，那么函数的返回值是 UI 描述(视图)。或者，正如 React 团队所说：“组件是状态机”。属性和状态都可以增强视图，但它们用于不同的目的(参见 4.3 节)。

要使用状态，可以使用名称访问它们。该名称是 `this.state` 对象的属性(又称为对象键或对象属性)，例如 `this.state.autocompleteMatches` 或 `this.state.inputFieldValue`。

**注意：**通常来说，状态是指组件中 `this.state` 对象的属性。根据上下文，状态有时可以指代单个属性(例如 `this.state.inputFieldValue`)，有时也可指代单个组件中 `state` 对象的多个属性。

状态数据通常用于在视图中显示动态信息，以便增强视图的渲染。回到自动完成示例中，状态在发送给服务器端的 XHR 请求响应中被修改，这是由用户在表单中输入触发的。当视图中使用的状态发生变化时，**React** 会保留视图的最新状态。实质上，当状态发生变化时，只有相应的部分会发生变化(单个元素或只是单个元素的属性值)。

DOM 中的其他内容将保持不变。这是可能的，因为有虚拟 DOM 的存在(参见 1.1.1 节)，它被 **React** 用于在协调过程中确定增量。这就是为什么能以声明式编写代码的原因，**React** 为你实现了背后的一切。对于视图更新的步骤及其如何发生，将在第 5 章中讨论。

**React** 开发人员使用状态来生成新的 UI。组件属性(`this.props`)、常规变量(`inputValue`)和类属性(`this.inputValue`)则不会这样，因为当更改其值时(在当前组件的上下文中)，不会触发视图更新。例如，以下反模式显示，如果更改除状态以外的任何值，视图将不会被更新：

```
// Anti-pattern: Stay away from it!
let inputValue = 'Texas'
class Autocomplete extends React.Component {
  updateValues() {
    this.props.inputValue = 'California'
    inputValue = 'California'
    this.inputValue = 'California'
  }
  render() {
    return (
      <div>
        {this.props.inputValue}
        {inputValue}
        {this.inputValue}
      </div>
    )
  }
}
```

← 用户交互时触发(键盘输入)

接下来，你将了解如何使用 **React** 组件状态。

**注意：**如前所述(复习是学习技巧之母)，如果从父组件传递新值，属性将会更新视图。这样会创建正在使用的组件的新实例。在给定组件的上下文中，像 `this.props.inputValue = 'California'` 这样更改属性将不会更新视图。

## 4.2 使用状态

为了使用状态，需要知道如何访问、更新它们并为它们设置初始值。下面从访问 **React** 组件的状态开始。

### 4.2.1 访问状态

`state` 对象是组件的属性对象，可以通过 `this` 引用来访问，例如 `this.state.name`。你应该还记得，可以在带有花括号(`{}`)的 **JSX** 中访问和输出变量。同样，也可以在 `render()` 方法中渲

染 `this.state`(和其他任意变量或自定义组件类属性一样); 例如 `{this.state.inputFieldValue}`。这种语法类似于通过 `this.props.name` 来访问属性。

使用目前所学知识, 我们来实现一个时钟, 如图 4.3 所示。目标是拥有一个自包含的组件类, 任何人都可以在应用中导入和使用它, 而不必费尽周折。时钟必须渲染当前时间。

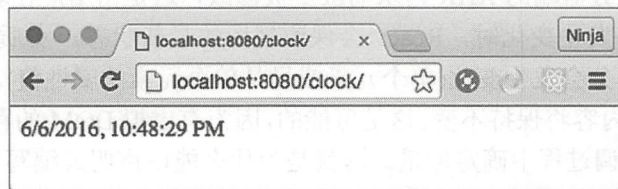


图 4.3 时钟组件以数字格式展示当前时间并且会按秒更新

时钟项目的结构如下:

```
/clock
  index.html
  /jsx
    script.jsx
    clock.jsx
  /js
    script.js
    clock.js
    react.js
    react-dom.js
```

我使用 Babel CLI 的 `watch(-w)` 和目录参数 `(-d)`, 把 `clock/jsx` 目录下所有的 JSX 文件编译到目标文件夹 `clock/js` 中, 并且会监听文件的变化, 实时编译。此外, 将命令保存为父目录 `ch04` 下 `package.json` 文件中的 `npm` 脚本, 以便可以在 `ch04` 目录下直接运行 `npm run build-clock`:

```
"scripts": {
  "build-clock": "./node_modules/.bin/babel clock/jsx -d clock/js -w"
},
```

显然, 时间在流逝(无论好坏)。因此, 需要更新视图——可以使用状态来实现。将其命名为 `currentTime`, 并尝试渲染状态, 参见代码清单 4.1。

#### 代码清单 4.1 在 JSX 中渲染状态

```
class Clock extends React.Component {
  render() {
    return <div>{this.state.currentTime}</div>
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('content')
)
```

你会得到如下报错: `Uncaught TypeError: Cannot read property 'currentTime' of null`。通



常, JavaScript 的错误消息非常鸡肋。这种还好, 因为至少 JavaScript 为你提供了一条有用的消息。这条消息意味着 `currentTime` 的值还不存在。与属性不同, 状态不是在父组件上设置的。也不能在 `render()` 方法中调用 `setState`, 因为这将导致死循环(`setState`→`render`→`setState`...), 这种情况下, React 会抛出错误。

## 4.2.2 设置初始状态

迄今为止, 你已经看到在 `render()` 方法中使用状态数据之前, 必须对其进行初始化。要设置初始状态, 可以在 ES6 `React.Component` 类语法的构造函数中使用 `this.state`。别忘了使用属性调用 `super()` 方法; 否则父组件(`React.Component`)中的逻辑将不会起作用:

```
class MyFancyComponent extends React.Component {
  constructor(props) {
    super(props)
    this.state = {...}
  }
  render() {
    ...
  }
}
```

还可以在设置初始状态时添加其他逻辑。例如, 可以使用 `new Date()` 设置 `currentTime` 的值。甚至可以使用 `toLocaleString()` 来获取用户所在地理位置的正确日期和时间格式, 如代码清单 4.2 所示(ch04/clock)。

代码清单4.2 Clock组件构造函数

```
class Clock extends React.Component {
  constructor(props) {
    super(props)
    this.state = {currentTime: (new Date()).toLocaleString()}
  }
  ...
}
```

`this.state` 的值必须是一个对象。此处不会详细介绍 ES6 的 `constructor()` 函数, 请参阅附录 E 和 ES6 备忘录: <https://github.com/azat-co/cheatsheets/tree/master/es6>。重要的是, 和其他 OOP 语言一样, `constructor()` 函数会在创建这个类的实例时被调用。构造函数的名字必须是 `constructor`, 可以将此视为 ES6 中的约定。此外, 如果创建 `constructor()` 函数, 那么几乎总是需要调用内部的 `super()` 方法; 否则, 将不会执行父类的构造函数。另外, 如果不定义 `constructor()` 方法, 那么 `super()` 方法会被默认调用。

### 类属性

希望 TC39 技术委员会(ECMAScript 规范的制定者)会在 ECMAScript 的未来版本中添加类属性语法。这样, 我们不仅可以在构造函数中设置 `state`, 还可以在类的主体中设置 `state`:

```
class Clock extends React.Component {
```

```
state = {  
  ...  
}  
}
```

类字段/类属性的提议可参考 <https://github.com/jeffmo/es-class-fields-and-static-properties>, 这份提议已经存在很多年了, 但是在本书撰写时(2017 年 3 月), 它还只是一份 stage 2 提案(stage 4 意味着最终的标准), 这表明它在浏览器中并没有得到广泛实现。也就是说, 这个功能无法正常工作(在本书撰写时, 还没有任何浏览器支持该特性)。

最可能的情况是, 将不得不使用编译器(例如 Babel、Traceur 或 TypeScript)来确保代码在所有浏览器中都可以正常工作。可以在 ECMAScript 兼容表(<http://kangax.github.io>)中查看类属性的当前兼容性。如果需要该特性, 请使用 Babel 预设 ES.Next。

这里, `currentTime` 是随意命名的, 在访问和更新这个状态时, 需要使用相同的名称。可以随意命名状态, 只需要稍后使用相同的名称引用它即可。

`state` 对象支持嵌套对象或数组。下面这个示例为状态添加了一个我个人书籍的数组:

```
class Content extends React.Component {  
  constructor(props) {  
    super(props)  
    this.state = {  
      githubName: 'azat-co',  
      books: [  
        'pro express.js',  
        'practical node.js',  
        'rapid prototyping with js'  
      ]  
    }  
  }  
  render() {  
    ...  
  }  
}
```

`constructor()` 方法只在从该类创建 React 元素时被调用一次。这样, 便可在 `constructor()` 方法中通过直接使用 `this.state` 来设置初始值。要避免在其他地方直接使用 `this.state=...` 设置和更新状态, 因为这样会导致意想不到的后果。

注意: 使用 React 的 `createClass()` 方法定义组件时, 需要使用 `getInitialState()` 方法。有关 `createClass` 和 ES5 中示例的更多信息, 请参阅 2.2 节中的小贴士“ES6+/ES2015+和 React”。

这只会给定一个初始值, 这个值很快就会过期, 例如, 在 1 秒钟后。没有显示当前时间的时钟是没有意义的, 幸运的是, 有一种方法可以更新状态。



### 4.2.3 更新状态

你将使用类方法 `this.setState(data, callback)` 来改变状态。当此方法被调用时，React 将 `data` 和当前状态合并，然后调用 `render()` 方法。之后，React 会调用 `callback`。

为 `setState()` 方法添加 `callback` 参数非常重要，因为该方法可以被异步调用。如果依赖新的状态，可以使用回调来确保这个新的状态可用。

如果需要依赖新的状态，但没有等待 `setState()` 方法执行结束，这等同于同步地使用异步操作，那么当依赖新的状态值时，可能会产生 Bug，因为状态值仍然是旧的。

迄今为止，你已经从状态中渲染了时间，也设置了初始状态，但是还需要每隔一秒钟就去更新时间，对吧？为此，可以使用浏览器计时器函数 `setInterval()` (<http://mng.bz/P2d6>)，它每隔 `n` 毫秒就会执行状态更新。几乎所有现代浏览器中都实现了 `setInterval()` 方法，这意味着可以在没有任何库的情况下使用它，如下所示：

```
setInterval(()=>{
  console.log('Updating time...')
  this.setState({
    currentTime: (new Date()).toLocaleString()
  })
}, 1000)
```

要启动时钟程序，需要调用一次 `setInterval()`。可以通过创建 `launchClock()` 方法来做到这一点。你将在构造函数中调用 `launchClock()`。最终的时钟代码如代码清单 4.3 (ch04/clock/jsx/clock.jsx) 所示。

代码清单4.3 使用状态实现时钟

```
class Clock extends React.Component {
  constructor(props) {
    super(props)
    this.launchClock()
    this.state = {
      currentTime: (new Date()).toLocaleString()
    }
  }
  launchClock() {
    setInterval(()=>{
      console.log('Updating time...')
      this.setState({
        currentTime: (new Date()).toLocaleString()
      })
    }, 1000)
  }
  render() {
    console.log('Rendering Clock...')
    return <div>{this.state.currentTime}</div>
  }
}
```

每隔一秒钟使用  
当前时间更新状态

触发 launchClock()

设置初始状态为  
当前时间

渲染状态

可以在任意地方使用 `setState()`，而不仅仅是在 `launchClock()` 方法中(在构造函数中调用)。通常，`setState()` 会在事件处理程序中，或者在接收新数据或数据更新的回调函数中调用。



**提示：**在代码中使用 `this.state.name='new name'` 改变状态值不会有任何好处。这不会触发你想要的重新渲染和可能发生的真正的 DOM 更新。在大多数情况下，不使用 `setState()`，而是直接改变状态是一种反模式，应该避免这么做。

需要注意，`setState()` 只更新传递给它的状态(部分或合并，而不是完全替换)。它不会每次都替换整个 `state` 对象。因此，如果有三个状态属性，那么只改变其中一个，另外两个并不会发生变化。在下面的示例中，`userEmail` 和 `userId` 将保持不变：

```
constructor(props) {  
  super(props)  
  this.state = {  
    userName: 'Azat Mardan',  
    userEmail: 'hi@azat.co',  
    userId: 3967  
  }  
}  
  
updateValues() {  
  this.setState({userName: 'Azat'})  
}
```

如果计划更新所有三个状态，那么需要显式地把所有状态的新值都传递给 `setState()` (在旧版本的 React 代码中，仍然可以看到另一个方法 `this.replaceState()`<sup>1</sup>，但它已被弃用并且不再被支持。正如从名称中可以猜测的那样，它可以用其参数替换整个 `state` 对象)。

请记住，`setState()` 会触发 `render()`，这在大多数情况下都有效。在某些场景中，代码依赖外部数据，可以使用 `this.forceUpdate()` 触发重新渲染。但这是一种不好的实践，应当避免。因为依赖外部数据而不是状态会使组件更加脆弱，并导致和外部因素紧密耦合。

如前所述，可以使用 `this.state` 来访问 `state` 对象，在 JSX 中使用花括号(`{}`)来输出状态值。因此，可以使用 `{this.state.NAME}` 在视图(即 `render()` 函数的 `return` 语句)中声明状态属性。

当在视图中(例如，在 `if/else` 语句中)使用状态数据，并使用新值调用 `setState()` 时，React 魔法将发生作用。React 为你更新必要的 HTML。可以在开发者工具的控制台中看到这一切。应该会先显示“Updating...”，然后显示“Rendering...”。最重要的是，只有绝对需要更新的 DOM 元素才会受影响。

### 在 JavaScript 中绑定 this

在 JavaScript 中，`this` 的值取决于在哪个函数中调用它。为了确保 `this` 指向组件类，需要将函数绑定到适当的上下文(这里是你的组件类)。

如果使用 ES6+/ES2015+，正如我在本书中所做的，可以使用箭头函数语法来创建一个自动绑定的函数：

1 <https://github.com/facebook/react/issues/3236>

```
setInterval(()=>{
  this.setState({
    currentTime: (new Date()).toLocaleString()
  })
}, 1000)
```

自动绑定的意思是：用箭头创建的函数内的 `this` 值是当前的 `this` 值，这里指的是 `Clock`。

手动绑定的方法是，在闭包中使用 `bind(this)` 方法：

```
function() {...}.bind(this)
```

看起来和下面的时钟程序一样：

```
setInterval(function(){
  this.setState({
    currentTime: (new Date()).toLocaleString()
  })
}.bind(this), 1000)
```

这种行为并非 React 独有。`this` 关键字在函数的闭包中会发生变化，因此需要一些绑定。当然，也可以保存上下文(`this`)的值，以便稍后可以使用。

通常，你会看到像 `self`、`that` 和 `_this` 这样的变量用来保存原始 `this` 的值。你可能见过下面这样的语句：

```
var that = this
var _this = this
var self = this
```

这种方法很直接：创建一个变量并在闭包中使用它，而不是使用 `this`。新变量不是一个副本，而是对原始 `this` 值的引用。`setInterval()` 如下：

```
var _this = this
setInterval(function(){
  _this.setState({
    currentTime: (new Date()).toLocaleString()
  })
}, 1000)
```

时钟现在已经开始工作了，如图 4.4 所示。

在继续之前还有一件事。可以看到 React 如何重用相同的 DOM 元素 `<div>`，且只改变它内部的文本。继续使用开发者工具修改这个元素的 CSS。添加使文本变蓝色的 `color:blue` 样式，如图 4.5 所示。这里创建了内联样式而不是类。当时间持续更新时，元素及其新的内联样式(蓝色)被保留了下来。

React 只会更新 `innerHTML`(第二个 `<div>` 容器的内容)。`<div>` 元素本身和页面上的其他元素一样，被完整、整齐地保留了下来。



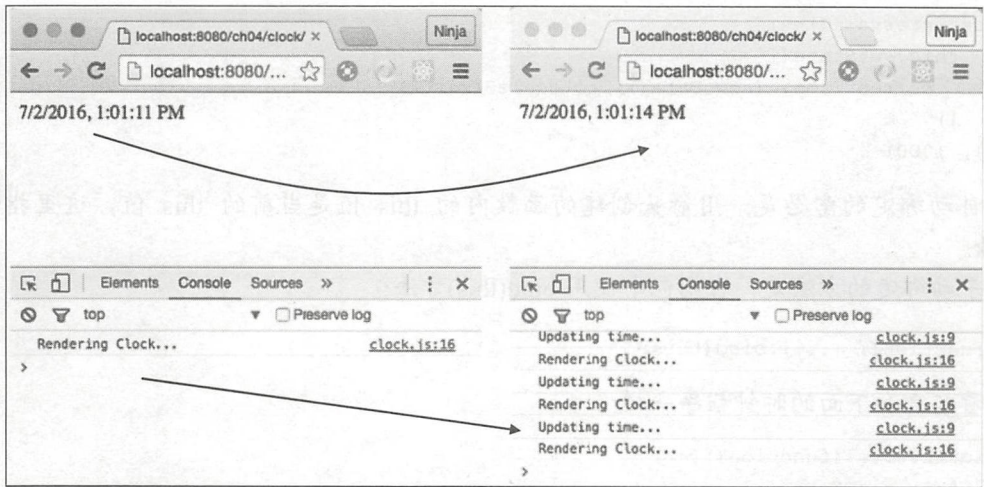


图 4.4 时间在流逝

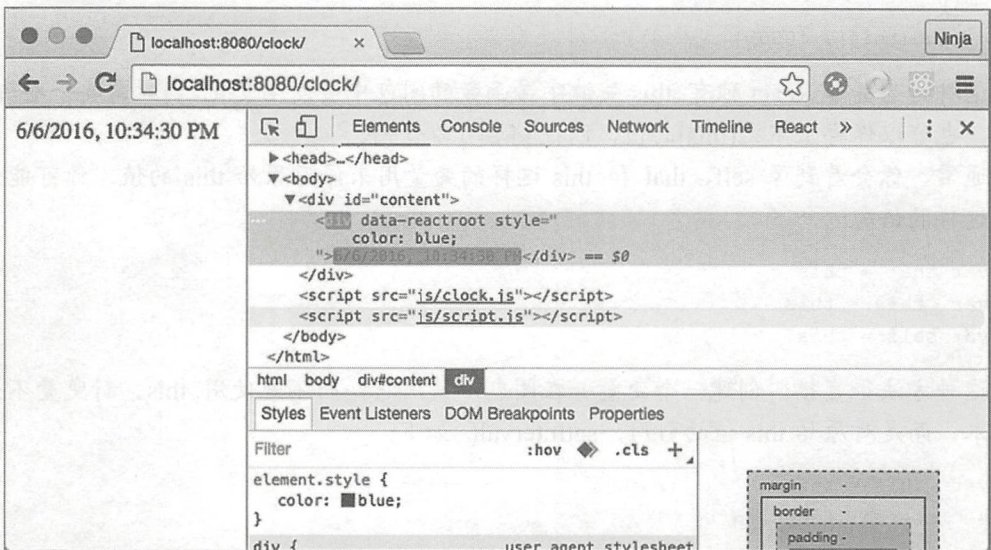


图 4.5 React 仅仅更新了时间文本，并未更新<div>元素(我手动添加了 color:blue 样式，它被一直保留了下来)

## 4.3 状态和属性

状态和属性都是类的特性，这意味着它们就是 `this.state` 和 `this.props`。这是两者唯一的相似点。状态和属性之间最主要的区别之一是：前者可变，后者不可变。

状态和属性的另一区别是：属性从父组件传递，状态在组件内部而非父组件中定义。这里的设计哲学是：属性只能从父组件变更，而不能从组件本身变更。因此，属性决定了创建时的视图，它们仍然是静态的(不会改变)。另一方面，状态随着状态对象的设置和更新而改变。

属性和状态的设计目的不同，但它们都可以当成组件类的属性来访问，并且都可以帮助组合组件呈现不同的视图。在组件的生命周期中(参见第 5 章)，状态和属性有一些不同。



把属性和状态想象成函数的输入，但会输出不同的视图。因此，每一组不同的状态和属性都会有不同的 UI(视图)，如图 4.6 所示。

并非所有组件都需要状态。下一节你将看到如何使用无状态组件的属性。

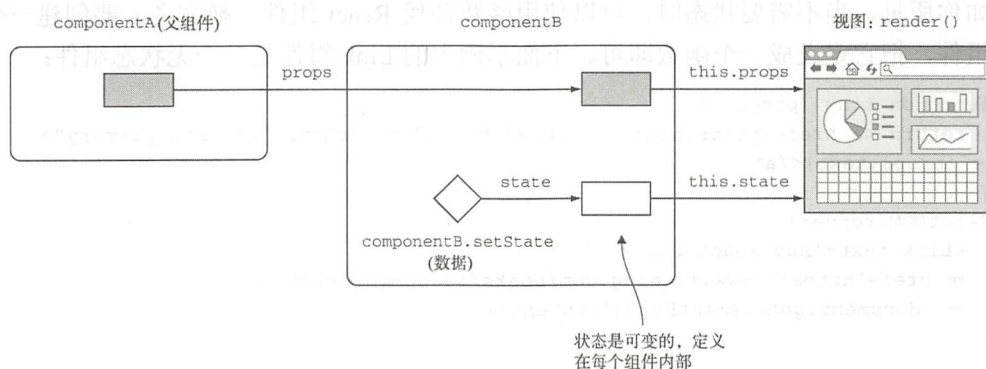


图 4.6 新的属性和状态值会改变 UI。新的属性值来自父组件，新的状态值来自组件

## 4.4 无状态组件

无状态组件没有状态或任何 React 生命周期事件/方法(参见第 5 章)。无状态组件的设计的目的只是渲染视图。它唯一能做的就是使用属性做一些事情。它其实是一个输入为属性、输出为 UI 元素的简单函数。

无状态组件的优点是可预测性，因为唯一输入决定了唯一输出。可预测性意味着更容易理解、维护和调试。事实上，没有状态是最理想的 React 实践——使用的无状态组件越多、有状态组件越少，效果就越好。

本书前 3 章编写了很多无状态组件。例如，Hello World 就是一个无状态组件(ch03/hello-js-world-jsx/jsx/script.jsx)，参见代码清单 4.4。

代码清单4.4 无状态Hello World

```
class HelloWorld extends React.Component {  
  render() {  
    return <h1 {...this.props}>Hello {this.props.frameworkName} world!!!</h1>  
  }  
}
```

React 为无状态组件提供了一种简单的函数风格语法，使用此语法可以创建一个带属性参数的函数并返回视图。无状态组件的渲染和其他组件如出一辙，例如，HelloWorld 组件可以改写成一个函数并返回<h1>：

```
const HelloWorld = function(props){  
  return <h1 {...props}>Hello {props.frameworkName} world!!!</h1>  
}
```

还可以使用 ES6+/ES2015+箭头函数来创建无状态组件。下面的代码段和上面的写法类似(return 也可以省略，但我个人喜欢保留它)：

```
const HelloWorld = (props) => {
  return <h1 {...props}>Hello {props.frameworkName} world!!!</h1>
}
```

如你所见，当不需要状态时，可以使用函数替代 React 组件。换言之，要创建一个无状态组件，把它定义成一个函数即可。下面示例中的 Link 组件是一个无状态组件：

```
function Link (props) {
  return <a href={props.href} target="_blank" className="btn btn-primary">
    {props.text}</a>
}
ReactDOM.render(
  <Link text='Buy React Quickly'
    href='https://www.manning.com/books/react-quickly'/>,
  document.getElementById('content')
)
```

尽管不需要自动绑定，但仍可使用箭头函数语法简化代码(仅有一条语句时，可写成一行)：

```
const Link = props => <a href={props.href}
  target="_blank"
  className="btn btn-primary">
  {props.text}
</a>
```

也可以在箭头函数中使用花括号({})、显式的 return 和圆括号(( ))，使代码更具可读性：

```
const Link = (props) => {
  return (
    <a href={props.href}
      target="_blank"
      className="btn btn-primary">
        {props.text}
    </a>
  )
}
```

在无状态组件中，不能拥有状态，但可以有两个属性：propTypes 和 defaultProps(分别参见 8.1 和 8.2 节)。可以在组件对象上设置它们。顺便提一下，如果是单行元素，可以没有圆括号：

```
function Link (props) {
  return <a href={props.href}
    target="_blank"
    className="btn btn-primary">
    {props.text}
  </a>
}
Link.propTypes = {...}
Link.defaultProps = {...}
```

无状态组件(函数)的引用(refs)也无法使用<sup>2</sup>。如果想使用,可将无状态组件包装在普通的 React 组件中。refs 更多信息参见 7.2.3 节。

## 4.5 有状态组件和无状态组件

为什么要使用无状态组件?当需要做的事情只是渲染一些 HTML 而无须创建支持实例和生命周期的组件时,它们更具声明性且运行得更好。基本上,当只需要将一些属性和元素组合到 HTML 中时,无状态组件可以减少冗余,并提供更好的语法和简单性。

我的建议,同时也是 React 官方团队推荐的最佳实践是:尽可能使用无状态组件替代普通的 React 组件。但是,如你所见,在时钟示例中,这并非总是可行的,有时候不得不使用状态。因此,在顶层结构上会有一些有状态组件来处理 UI 状态、交互和其他应用逻辑(例如从远程服务器加载数据)。

不要认为无状态组件就必须是静态的。给它们传递不同的属性,就可以改变它们的展现效果。来看一个示例:它将 Clock 重构为三个组件——一个有状态的时钟,包含状态和更新逻辑;还有两个仅仅输出时间的无状态组件——DigitalDisplay 和 AnalogDisplay(使用不同的方法)。目标效果类似于图 4.7,很漂亮,不是吗?

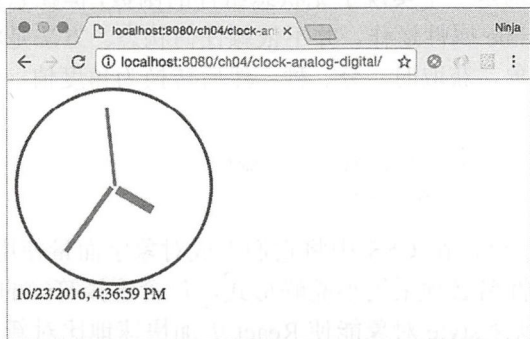


图 4.7 时钟程序使用两种方式显示时间:模拟形式和数字形式

项目的目录结构如下:

```
/clock-analog-digital
  /jsx
    analog-display.jsx
    clock.jsx
    digital-display.jsx
    script.jsx
  /js
    analog-display.js
    clock.js
    digital-display.js
    script.js
    react.js
    react-dom.js
  index.html
```

2 “React stateless component this.refs..value?”, <http://mng.bz/Eb91>



时钟程序渲染了两个子元素，并使用状态 `currentTime` 作为 `time` 属性的值传递给子组件，参见代码清单 4.5。父组件的状态成了子组件的属性。

代码清单4.5 把状态传递给子组件

```
...
render() {
  console.log('Rendering...')
  return <div>
    <AnalogDisplay time={this.state.currentTime}/>
    <DigitalDisplay time={this.state.currentTime}/>
  </div>
}
```

现在，需要创建 `DigitalDisplay`。它很简单，一个带有属性参数的函数通过属性参数 (`props.time`) 显示时间值，如代码清单 4.6 所示 (`ch04/clock-analog-digital/jsx/digital-display.jsx`)。

代码清单4.6 无状态数字显示组件

```
const DigitalDisplay = function(props) {
  return <div>{props.time}</div>
}
```

`AnalogDisplay` 同样也是一个实现了无状态组件的函数，但在它内部有一些酷炫的动画来旋转指针。动画基于 `time` 属性运作，并不依赖任何状态。方法是将时间设置成一个字符串，把它转换为 `Date` 对象，获取时、分、秒，然后转换为角度值。例如，下面展示了如何将秒数转换为角度值：

```
let date = new Date('1/9/2007, 9:46:15 AM')
console.log((date.getSeconds()/60)*360 ) // 90
```

一旦得到角度值，就可以在 CSS 中将它们当成对象字面量使用。和常规 CSS 相比，React CSS 中的 `style` 属性名必须采用小驼峰形式，包含连接符(-)的属性名在 JavaScript 中是非法的。之前提及，包含 `style` 对象能使 React 更加快速地对新老元素之间的差异。更多关于 React `style` 和 CSS 的内容，参见 3.4.3 小节。

代码清单 4.7 展示了带有 CSS 的无状态模拟显示组件，它使用 `time` 属性作为输出值 (`ch04 /clock-digital/jsx /analog-display.jsx`)。

代码清单4.7 无状态模拟显示组件

```
const AnalogDisplay = function AnalogDisplay(props) {
  let date = new Date(props.time)
  let dialStyle = {
    position: 'relative',
    top: 0,
    left: 0,
    width: 200,
    height: 200,
    borderRadius: 20000,
    borderStyle: 'solid',
    borderColor: 'black'
  }
  let secondHandStyle = {
    position: 'relative',
    top: 100,
    left: 100,
```

← 把字符串形式的日期转换成对象，以便稍后对它进行解析

← 在<div>上使用borderRadius (常规CSS中的border-radius), 设置一个非常高的数值，使其变成圆形

```
border: '1px solid red',
width: '40%',
height: 1,
transform: 'rotate(' + ((date.getSeconds()/60)*360 - 90 )
    .toString() + 'deg)',
transformOrigin: '0% 0%',
backgroundColor: 'red'
}
let minuteHandStyle = {
  position: 'relative',
  top: 100,
  left: 100,
  border: '1px solid grey',
  width: '40%',
  height: 3,
  transform: 'rotate(' + ((date.getMinutes()/60)*360 - 90 )
    .toString() + 'deg)',
  transformOrigin: '0% 0%',
  backgroundColor: 'grey'
}
let hourHandStyle = {
  position: 'relative',
  top: 92,
  left: 106,
  border: '1px solid grey',
  width: '20%',
  height: 7,
  transform: 'rotate(' + ((date.getHours()/12)*360 - 90 ).toString() + 'deg)',
  transformOrigin: '0% 0%',
  backgroundColor: 'grey'
}
return <div>
  <div style={dialStyle}>
    <div style={secondHandStyle}/>
    <div style={minuteHandStyle}/>
    <div style={hourHandStyle}/>
  </div>
</div>
}
```

计算角度，值并且让秒针相对  
初始位置旋转，相当于将角度  
值减去90

使用transformOrigin设置  
旋转中心点的偏移量

以相对于时钟刻度盘  
(大圆)的方式渲染容器

如果为 Chrome 和 Firefox 浏览器安装了 React 开发者工具(可以从 <http://mng.bz/mt5P> 和 <http://mng.bz/DANq> 下载)。在开发者工具(或 Firefox 中的 analog)中打开 React 面板。数据显示: <Clock>元素拥有两个子元素,如图 4.8 所示。React 开发者工具会告诉你组件的名称,以及状态和 currentTime(当前时间)。这个调试工具简直太棒了!

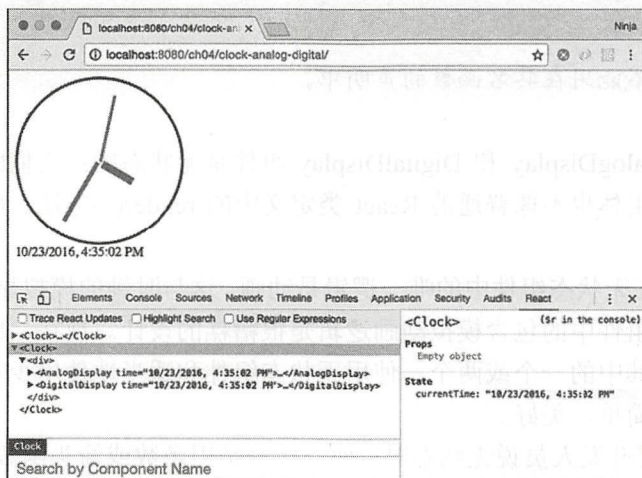


图 4.8 React 开发者工具 v0.15.4 显示了两个组件



请注意，本例使用一个匿名表达式并将之保存到声明为 `const` 的变量中。另一种方法是使用具名函数声明语法：

```
function AnalogDisplay(props) {...}
```

也可以使用变量来引用具名函数声明：

```
const AnalogDisplay = function AnalogDisplay(props) {...}
```

### JavaScript 中的函数声明

在 JavaScript 中，有几种方法可以定义函数。

使用匿名函数表达式(通常作为回调)：

```
function() { return 'howdy' }
```

创建 IIFE：

```
(function() {  
  return('howdy')  
})();
```

使用变量引用匿名函数表达式：

```
let sayHelloInMandarin = function() { return 'nǐ haō' }
```

下面是一个使用变量引用的具名函数表达式：

```
function sayHelloInTatar() { return 'sälam' }
```

And this is a named or hoisted function expression referenced in a variable:

```
let sayHelloInSpanish = function digaHolaEnEspanol() { return 'hola' }
```

最后，可以使用立即执行函数表达式：

```
(function sayHelloInTexan() {  
  return('howdy')  
})();
```

箭头函数语法不能用在具名函数的声明中。

如你所见，`AnalogDisplay` 和 `DigitalDisplay` 组件是无状态的：它们既没有状态，也没有任何方法。函数主体也不像普通的 React 类定义中的 `render()` 方法。所有逻辑和状态都在 `Clock` 组件中。

与此相反，放在无状态组件中的唯一逻辑是动画，这与时钟的模拟显示逻辑密切相关。很明显，在 `Clock` 组件中的包含模拟动画逻辑是很糟糕的设计。现在，已有两个组件，可以从 `Clock` 中渲染其中的一个或两个。使用无状态组件和适当地使用少量有状态组件，可以使设计更灵活、简单、美好。

通常，当 React 开发人员说无状态时，他们指的是用函数或箭头函数语法创建的组件。使用类创建无状态组件虽然是可行的，但并不推荐，因为对其他人(或由你自己在 6 个月内)



而言, 添加状态太容易了。没有诱惑, 便没有办法使代码复杂化!

或许你想知道无状态组件是否可以有方法。很明显, 如果使用类, 它们可以有方法。但正如前面提及的, 大多数开发人员使用函数。虽然可以将方法附加到函数(也是 JavaScript 对象)上, 但并不优雅, 因为在函数中无法使用 `this`(`this` 的值不是组件, 而是 `window` 对象):

```
// Anti-pattern: Don't do this.
const DigitalDisplay = function(props) {
  return <div>{DigitalDisplay.locale(props.time)}</div>
}
DigitalDisplay.locale = (time)=>{
  return (new Date(time)).toLocaleString('EU')
}
```

如果需要执行一些与视图相关的逻辑, 可以在无状态组件中创建一个新的函数:

```
// Good pattern
const DigitalDisplay = function(props) {
  const locale = time => (new Date(time)).toLocaleString('EU')
  return <div>{locale(props.time)}</div>
}
```

请保持无状态组件的简单性: 没有状态, 没有方法。特别是, 不要调用外部的方法或函数, 因为这样会打破组件的可预测性(并且违背了纯粹性的概念)。

## 4.6 测验

1. 在组件方法中可以使用以下哪种语法设置状态? `this.setState(a)`、`this.state = a`、`this.a = a`。
2. 如果想要更新渲染过程, 一般的做法是改变组件的属性, 例如 `this.props.a = 100`, 这么理解正确还是错误?
3. 状态是可变的, 属性是不可变的, 这么理解正确还是错误?
4. 无状态组件可以使用函数来实现, 这么理解正确还是错误?
5. 当创建元素时, 如何定义初始状态变量? `setState()`、`initialState()`、在构造函数中使用 `this.state = ...` 还是 `setInitialState()`?

## 4.7 小结

- 状态是可变的, 属性是不可变的。
- `getInitialState` 允许组件有初始 `state` 对象。
- `this.setstate` 仅更新传递给它的属性, 而不是更新所有 `state` 对象属性。
- `{}` 是一种渲染变量和在 `JSX` 代码中执行 `JavaScript` 的方法。
- `this.state.NAME` 是访问状态变量的方法。
- 无状态组件是 `React` 创建组件的首选。

## 4.8 测验答案

1. `this.setState(a)`。因为除了在 `constructor()` 中，从不会直接给 `this.state` 赋值。`this.a` 不会对状态做任何处理，而只会创建一个实例属性。
2. 错误。在组件中修改属性不会触发重新渲染。
3. 正确。在组件中无法修改属性，只能从它的父组件设置。相反，状态只能在组件内部修改。
4. 正确。可以使用箭头函数或传统的 `function(){}` 来定义。
5. 在构造函数中使用 `this.state=...`，或者如果使用 `createClass()` 的话，还可以使用 `getInitialState()`。

# 第 5 章

## React 组件生命周期

### 本章内容：

- React 组件生命周期的全景视图
- 理解事件分类
- 定义事件
- 挂载、更新和卸载事件

第 2 章介绍了如何创建组件，但在某些情况下，需要对组件进行更细粒度的控制。例如，构建一个自定义的单选按钮组件，该组件可以根据屏幕宽度改变大小；或者构建一个菜单，需要通过发送 XHR 请求来从服务器获取信息。

一种方法是在实例化组件之前实现必要的逻辑，然后通过提供不同的属性重新创建组件。遗憾的是，这种方法无法创建自包含组件，因此也将失去基于组件的架构所带来的优势。

最好的方法是使用生命周期事件。通过挂载事件，可以将必要的逻辑注入组件。此外，还可以使用其他事件，通过提供特定逻辑使组件变得更加智能，以确定是否需要重新渲染视图(覆盖 React 默认算法)。

回到自定义单选按钮和菜单示例，可以在创建按钮组件时将事件监听器绑定到 window(onResize)上，并在组件被移除时解绑。当把 React 元素挂载到真实 DOM 上时，菜单可以从服务器获取数据。

下面就开始学习组件生命周期事件吧！



## 5.1 React 组件生命周期事件的全景视图

React 提供了一种基于生命周期事件(想想计算机编程中的钩子(<https://en.wikipedia.org/wiki/Hooking>))的方法来控制 and 自定义组件行为。这些事件可以归为以下几类:

- 挂载事件: 发生在 React 元素(组件类的实例)被绑定到真实 DOM 节点上时。
- 更新事件: 发生在 React 元素有新的属性或状态需要更新时。
- 卸载事件: 发生在 React 元素从 DOM 中卸载之时。

每一个 React 组件都有生命周期事件, 这些事件的触发时机取决于组件将要做或者已经做了什么。某些事件只会执行一次, 其他的可以执行多次。

生命周期事件允许通过实现自定义逻辑来增强组件的能力。还可以借助它们修改组件的行为: 例如, 决定何时重新渲染。这将提高性能, 因为不需要的操作被消除了。另一种用法是从后端获取数据或与 DOM 事件、其他前端类库集成。接下来让我们更深入地探索事件分类是如何运作的、它们所拥有的事件以及这些事件的执行顺序。

## 5.2 事件的分类

React 将所有组件生命周期事件定义成三类(如图 5.1 和本章后面的表 5.1 所示), 每个分类可以触发不同次数的事件:

- 挂载: 仅调用一次
- 更新: 调用多次
- 卸载: 仅调用一次

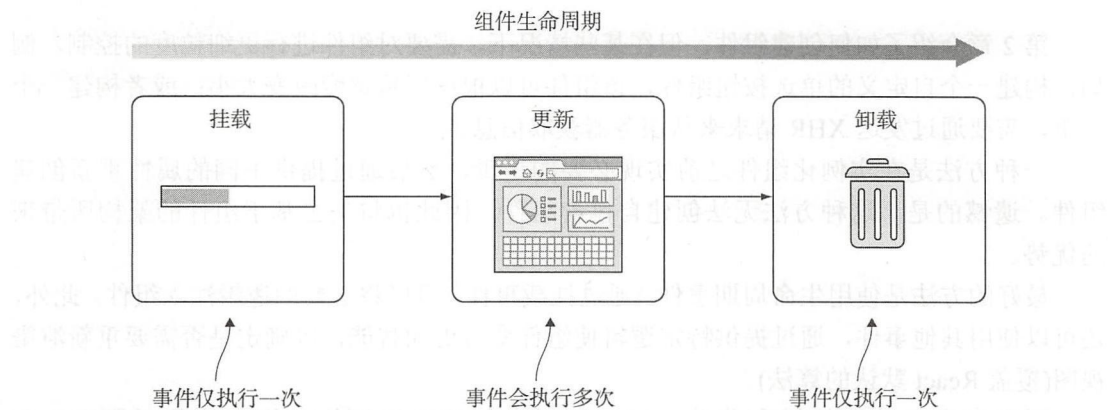


图 5.1 组件整个生命周期中生命周期事件的分类, 以及每类事件可以被调用的次数

除了生命周期事件之外, 我会把 `constructor()` 也囊括在内, 以便演示组件生命周期从开始到结束的执行顺序(更新可以发生多次):

- `constructor()`: 发生在元素创建时, 可以设置默认的属性(参见第 2 章)和初始状态(参见第 4 章)。
- 挂载
  - `componentWillMount()`: 发生在挂载到 DOM 之前。

- `componentDidMount()`：发生在挂载和渲染之后。
- 更新
  - `componentWillReceiveProps(nextProps)`：发生在组件即将接收属性时。
  - `shouldComponentUpdate(nextProps, nextState)`：通过判断何时需要更新、何时不需要更新、允许对组件的渲染进行优化。
  - `componentWillUpdate(nextProps, nextState)`：发生在组件将要更新之前。
  - `componentDidUpdate(prevProps, prevState)`：发生在组件更新完成之后。
- 卸载
  - `componentWillUnmount()`：允许在组件卸载之前解绑所有的事件监听器或者其他清理操作。

通常，当事件触发时，事件的名称已经向开发者明示了一切。例如，`componentDidUpdate()`会在组件更新完成后触发。其他情况下，有一些细微差别。表 5.1 展示了生命周期事件的顺序(从上至下)，以及某些事件如何依赖组件属性和状态的变化。

表 5.1 生命周期事件(以及它们与状态和属性的关系)

挂载	<code>constructor()</code> <code>componentWillMount()</code> <code>render()</code> <code>componentDidMount()</code>
更新组件属性	<code>componentWillReceiveProps()</code> <code>shouldComponentUpdate()</code> <code>componentWillUpdate()</code> <code>render()</code> <code>componentDidUpdate()</code>
更新组件状态	<code>shouldComponentUpdate()</code> <code>componentWillUpdate()</code> <code>render()</code> <code>componentDidUpdate()</code>
使用 <code>forceUpdate()</code> 更新	<code>componentWillUpdate()</code> <code>render()</code> <code>componentDidUpdate()</code> <code>componentWillUnmount()</code>

还有一个调用 `this.forceUpdate()` 的例子，其中的组件可能会被重新渲染。从名字可以猜出，它强制更新。可以在因为各种原因导致更新状态或属性不会触发期望的重新渲染时使用它。例如，这可能发生在当在 `render()` 中使用不属于状态和属性的数据，并且数据发生变更时，这种情形下需要手动触发更新。一般情况下(参照 React 官方团队的建议)，应该避免使用 `this.forceUpdate()` 方法(<http://mng.bz/v5sU>)，因为它使组件变得不纯(参见下面的小贴士)。接下来，我们实际定义一个事件来看看。

### 纯函数

在计算机科学中(不仅仅是在 React 中)，纯函数是下面这样的函数：

- 给定同样的输入，总是会得到同样的输出。



- 没有副作用(不改变外部状态)。
- 不依赖任何外部状态。

例如,有一个对输入的数值乘以 2 的纯函数—— $f(x)=2x$ ,在 JavaScript/Node 中表示为 `let f=(n)=>2*n`,如下所示:

```
let f = (n)=>2*n
console.log(f(7))
```

对数值翻倍的非纯函数的实现大致如下(对于单行箭头函数,加上花括号可以省略显式的 `return`):

```
let sharedStateNumber = 7
let double
let f = ()=> {double =2*sharedStateNumber}
f()
console.log(double)
```

纯函数是函数式编程(Functional Programming, FP)的基础,要旨是尽可能减少状态。开发人员(尤其是函数式编程人员)更偏好纯函数的主要原因是:使用纯函数可以减少共享状态,从而简化开发,将不同的逻辑分离开来。此外,测试也更加容易。对于 React 而言,你已经知道,拥有更多的无状态组件和更少的依赖项会更好,这就是为什么最佳实践是创建纯函数的原因。

在某些方面,FP 和 OOP 是相悖的(或者说 OOP 和 FP 是相悖的),FP 的粉丝们说 Fortran 和 Java 都是编程的死胡同,Lisp(以及 nowadays、Clojure 和 Elm)才是正道。这是一场令人着迷的辩论。就我个人而言,我更偏向函数式编程方法。

已有许多优秀的关于 FP 的书籍,因为这个概念已经存在几十年了。鉴于此,我不会讲太多细节,但是我非常想要学习关于 FP 的更多知识,因为它会让你成为一名更好的程序员,即使你从未打算在工作中使用它。

## 5.3 实现生命周期事件

要实现生命周期事件,可以在组件类中定义一个方法(参见 3.2.5 节)——这是 React 期望开发者能遵循的约定。React 会检查是否存在一个和事件同名的方法,如果发现存在,React 会调用该方法。否则,React 将继续执行正常的工作流程。显然,事件名称与 JavaScript 中的任何名称一样,都是大小写敏感的。

换言之,React 的底层逻辑是:在组件的生命周期中,如果它们被定义,就会被调用。例如,如果定义了 `componentDidMount()`,那么当这个组件类的元素被挂载时,React 将调用该方法。`componentDidMount()`属于表 5.1 中列出的挂载类别,它会在组件类每次实例化时被调用一次:

```
class Clock extends React.Component {
```



```
componentDidMount() {  
  ...  
}
```

如果没有定义 `componentDidMount()` 方法, 那么 React 不会执行该事件的任何代码。因此, 方法的名称必须与事件的名称匹配。我将在本章中交替使用术语事件、事件处理程序和方法。

正如从事件名中能够猜到的, `componentDidMount()` 方法将在组件插入 DOM 时被调用。这个方法被推荐用来放置一些和其他框架与库集成的代码, 以及向服务器端发送 XHR 请求。因为在生命周期的这个时间点, 组件已经存在于 DOM 中, 所有元素(包括子节点)都已经可以访问。

回到本章开头提出的问题: 调整大小, 从服务器端获取数据。首先, 可以在 `componentDidMount()` 中创建 `window.resize` 事件监听器。其次, 可以在该方法中发送一个 XHR 请求, 并且当收到服务器端响应时更新状态。

同样重要的是, `componentDidMount()` 在同构代码(在服务器和浏览器中使用相同的组件)中也非常有用。可以在该方法中添加仅浏览器端的逻辑, 并确保只会在浏览器中调用, 而不在服务器端调用。第 16 章中有更多的同构 JavaScript。

大多数开发人员通过查看示例可以达到最佳的学习效果, 出于这个原因, 我们考虑一个使用 `componentDidMount()` 将 DOM 信息打印到控制台的小示例。这是可行的, 因为这个事件会在所有渲染发生之后触发, 因此可以访问 DOM 元素。

为组件生命周期事件添加事件监听器很简单: 在组件类中定义方法即可。我们可以尝试添加 `componentWillMount()` 来作对比, 这个阶段元素的真实 DOM 还不存在。

DOM 节点信息是通过 React DOM 的工具函数 `ReactDOM.findDOMNode()` 获得的, 并在组件类中传递。注意, DOM 不使用驼峰命名法, 而是全大写。

```
class Content extends React.Component {  
  componentWillMount() {  
    console.log(ReactDOM.findDOMNode(this))  
  }  
  componentDidMount() {  
    console.dir(ReactDOM.findDOMNode(this))  
  }  
  render() {  
    return (  
      ...  
    )  
  }  
}
```

预期的DOM值为null

预期的DOM值为<div>

结果被输出到开发者工具的控制台中, 让人放心的是: `componentDidMount()` 在拥有真实 DOM 元素时被执行了, 如图 5.2 所示。

```
html  
null  
div
```

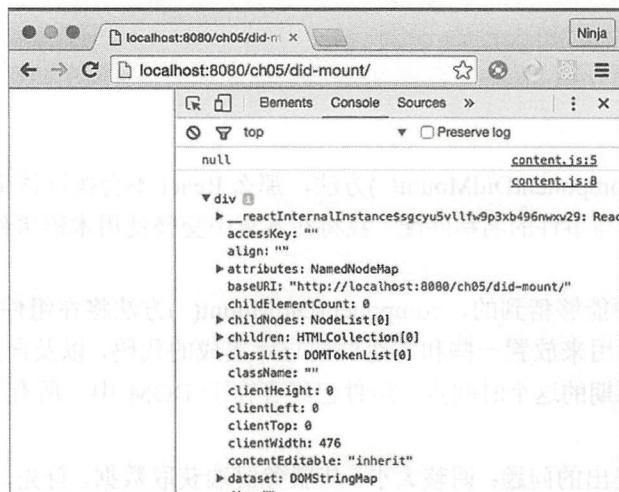


图 5.2 第二个日志显示了 DOM 节点，因为当元素被渲染，然后挂载到实际 DOM 时，`componentDidMount()` 被触发。因此，节点得以显示出来

## 5.4 执行所有事件

代码清单 5.1(ch05/logger/jsx/content.jsx)和代码清单 5.2(ch05/logger/jsx/logger.jsx)展示了如何一次性执行所有事件。现在，你需要知道它们就像类一样允许代码复用。这个 `logger` 组件在调试时非常有用，它显示了组件即将渲染和已经渲染时的所有事件、属性和状态。

代码清单5.1 渲染和更新logger组件三次

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.launchClock()
    this.state = {
      counter: 0,
      currentTime: (new Date()).toLocaleString()
    }
  }
  launchClock() {
    setInterval(() => {
      this.setState({
        counter: ++this.state.counter,
        currentTime: (new Date()).toLocaleString()
      })
    }, 1000)
  }
  render() {
    if (this.state.counter > 2) return
    return <Logger time="{this.state.currentTime}"></Logger>
  }
}
```

## 代码清单5.2 观察组件的生命周期事件

```
class Logger extends React.Component {
  constructor(props) {
    super(props)
    console.log('constructor')
  }
  componentWillMount() {
    console.log('componentWillMount is triggered')
  }
  componentDidMount(e) {
    console.log('componentDidMount is triggered')
    console.log('DOM node: ', ReactDOM.findDOMNode(this))
  }
  componentWillReceiveProps(newProps) {
    console.log('componentWillReceiveProps is triggered')
    console.log('new props: ', newProps)
  }
  shouldComponentUpdate(newProps, newState) {
    console.log('shouldComponentUpdate is triggered')
    console.log('new props: ', newProps)
    console.log('new state: ', newState)
    return true
  }
  componentWillUpdate(newProps, newState) {
    console.log('componentWillUpdate is triggered')
    console.log('new props: ', newProps)
    console.log('new state: ', newState)
  }
  componentDidUpdate(oldProps, oldState) {
    console.log('componentDidUpdate is triggered')
    console.log('new props: ', oldProps)
    console.log('old props: ', oldState)
  }
  componentWillUnmount() {
    console.log('componentWillUnmount')
  }
  render() {
    // console.log('rendering... Display')
    return (
      {this.props.time}
    )
  }
}
```

运行这个 Web 页面时，控制台会显示 Display 组件生命周期事件输出的日志。不要忘记



记打开浏览器控制台，因为所有的日志都输出在这里，如图 5.3 所示！

正如前文和图 5.3 中所示，挂载事件只会执行一次。可以在日志中清楚地看到这一点。在 Context 中的计数器到达 3 之后，render() 函数不再使用 Display 组件，组件将被卸载(参见图 5.4)。

既然已经了解了组件生命周期事件，现在可以在需要为组件实现逻辑时使用它们，比如获取数据。



图 5.3 logger 组件已经被挂载

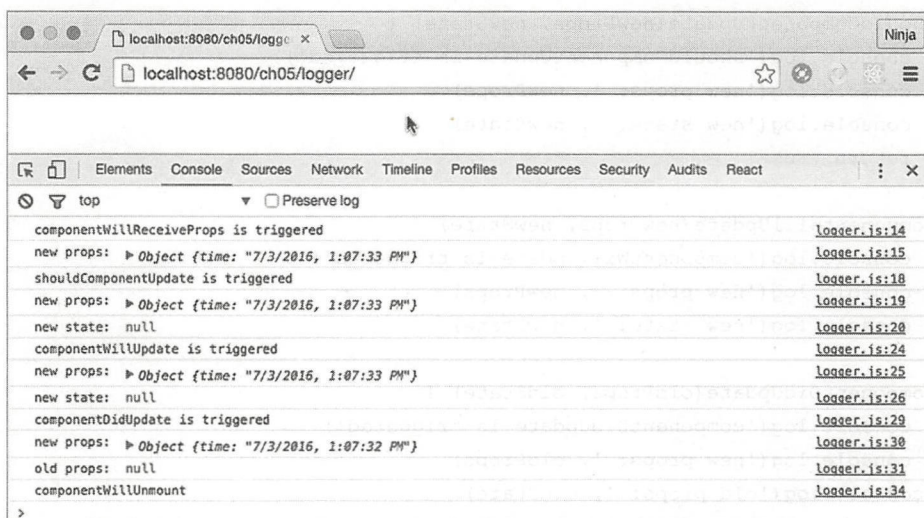


图 5.4 Content 组件在两秒钟之后会移除 logger 组件，因此，componentWillUnmount() 日志出现在删除之前是正确的

## 5.5 挂载事件

挂载事件主要是关于组件被渲染到真实 DOM 的过程。可以把挂载行为视为 React 元素要在 DOM 中看到自己，这通常发生在对组件使用 ReactDOM.render() 时，或用在其他高阶组件的 render() 方法中。组件将会被渲染到 DOM 中，挂载事件如下所示：

- componentWillMount(): React 知道该元素即将被渲染到真实 DOM 中。

- `componentDidMount()`: React 已经将 React 元素渲染到真实 DOM 中, 并且元素就是 DOM 节点。

`constructor()` 会在 `componentWillMount()` 之前执行。另外, React 会先执行渲染, 然后挂载元素(这里渲染的含义是调用组件类的 `render()` 方法, 而不是绘制 DOM)。 `ComponentWillMount()` 和 `componentDidMount()` 之间的事件参见表 5.1。

### 5.5.1 `componentWillMount()`

值得一提的是, `componentWillMount()` 在组件生命周期中仅调用一次。执行的时机正好在初始化渲染之前。

当在浏览器中调用 `ReactDOM.render()` 来渲染 React 元素时, 生命周期事件 `componentWillMount()` 将会被执行。可以将此看成即将把一个 React 元素绑定(或挂载)到真实 DOM 节点上。这些都发生在浏览器端: 也叫前端。

如果在服务器端(后端)渲染一个 React 组件(使用同构 JavaScript, 参见第 16 章), 会得到一个 HTML 字符串。即便服务器端没有 DOM, 也没有挂载的过程, 这个事件也会被调用。

第 4 章介绍了如何使用 `Date` 和 `setInterval()` 来更新 `currentTime` 状态。通过调用 `launchClock()`, 在 `constructor()` 中触发了一系列的更新。也可以在 `componentWillMount()` 中来做这些事情。

通常情况下, 状态更改会触发重新渲染, 对吗? 如果在 `componentWillMount()` 方法中使用 `setState()` 更新状态, 或者像在时钟程序中那样触发更新, 那么 `render()` 将得到更新后的状态。最好的情况是, 即使新的状态不同, 也不会执行重新渲染, 因为 `render()` 将会得到新的状态。换言之, 可以在 `componentWillMount()` 中调用 `setState()`。 `render()` 方法将会得到最新的值, 并且不会再执行额外的渲染。

### 5.5.2 `componentDidMount()`

相反, `componentDidMount()` 会在初始化渲染之后被调用。这个事件仅执行一次且仅在浏览器端执行, 在服务器端不会执行。当需要实现一些仅仅运行在浏览器端的方法时, 这会很方便, 例如发送 XHR 请求。

在这个生命周期事件中, 可以访问任何子组件的引用(例如, 访问对应的 DOM 元素)。注意, 子组件的 `componentDidMount()` 将在父组件该事件调用之前被调用。

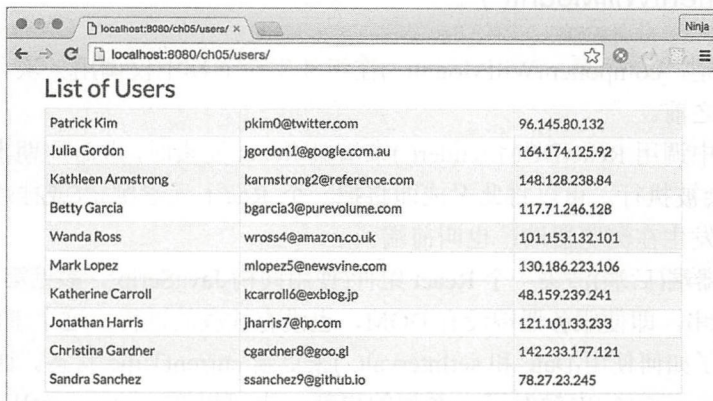
如前所述, `componentDidMount()` 事件是与其他 JavaScript 库集成的最佳场所。可以获取一个 JSON payload, 其中包含用户的信息列表。然后可以打印该信息, 使用 Twitter Bootstrap 表格来展现页面, 如图 5.5 所示。

项目结构如下:

```
/users
  /css
    bootstrap.css
  /js
    react.js
```



```
react-dom.js
script.js
- users.js
/jsx
script.jsx
users.jsx
index.html
real-user-data.json
```



Patrick Kim	pkim0@twitter.com	96.145.80.132
Julia Gordon	jgordon1@google.com.au	164.174.125.92
Kathleen Armstrong	karmstrong2@reference.com	148.128.238.84
Betty Garcia	bgarcia3@purevolume.com	117.71.246.128
Wanda Ross	wross4@amazon.co.uk	101.153.132.101
Mark Lopez	mlopez5@newsvine.com	130.186.223.106
Katherine Carroll	kcarroll6@exblog.jp	48.159.239.241
Jonathan Harris	jharris7@hp.com	121.101.33.233
Christina Gardner	cgardner8@goo.gl	142.233.177.121
Sandra Sanchez	ssanchez9@github.io	78.27.23.245

图 5.5 展现使用 Twitter Bootstrap 风格的用户列表(从数据存储中获取)

在该事件中, DOM 元素已经存在, 可以使用最新的 `fetch()` API 发送 XHR/AJAX 请求来获取数据:

```
fetch(this.props['data-url'])
  .then((response)=>response.json())
  .then((users)=>this.setState({users: users}))
```

### Fetch API

Fetch API(<http://mng.bz/mbMe>)允许以统一的方式使用 Promise 发送 XHR 请求。大多数现代浏览器都支持, 但是需要参考规范(<https://fetch.spec.whatwg.org>)和标准(<https://github.com/whatwg/fetch>)来找出需要支持的浏览器是否已经实现了 Fetch API。用法很简单, 传递一个 URL, 然后根据需要定义多个 `then` 语句:

```
fetch('http://node.university/api/credit_cards/')
  .then(function(response) {
    return response.blob()
  })
  .then(function(blob) {
    // Process blob
  })
  .catch(function(error) {
    console.log('A problem with your fetch operation: ' +
      error.message)
  })
```

如果你的浏览器不支持 `fetch()`, 可以对它进行 shim, 或者使用其他 HTTP 代理库, 例如





superagent(<https://github.com/visionmedia/super-agent>)、request(<https://github.com/request/request>)、axios(<https://github.com/mzabriskie/axios>)，甚至 jQuery 的 \$.ajax() (<http://api.jquery.com/jquery.ajax>) 或 \$.get()。

可以将获取请求放在 `componentDidMount()` 中。你可能会认为，将代码放在 `componentWillMount()` 中可以优化加载，但是有两个问题：如果从服务器获取数据比渲染完成的速度快，那么可能会触发未挂载元素的重新渲染，这可能会导致意外的结果；另外，如果打算在服务器端使用组件，那么 `componentWillMount()` 也会在那里触发。

现在来看看整个组件，在 `componentDidMount()` (`ch05/users/jsx/users.jsx`) 中进行获取操作。

#### 代码清单5.3 获取数据，然后展现在表格中

```
class Users extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      users: []
    }
  }
  componentDidMount() {
    fetch(this.props['data-url'])
      .then((response) => response.json())
      .then((users) => this.setState({users: users}))
  }
  render() {
    return <div className="container">
      <h1>List of Users</h1>
      <table className="table-striped table-condensed table table-bordered
        ➡ table-hover">
        <tbody>{this.state.users.map((user) =>
          <tr key={user.id}>
            <td>{user.first_name} {user.last_name}</td>
            <td> {user.email}</td>
            <td> {user.ip_address}</td>
          </tr>)}
        </tbody>
      </table>
    </div>
  }
}
```

使用一个空数组初始化users 的状态

执行GET XHR请求，  
使用来自属性的URL获取  
用户数据

从响应中获取用户信息，  
并将其设置到状态中

迭代users以便创建表格行

注意，在 `constructor()` 中，`users` 被设置为空数组。这就需要稍后在 `render()` 中校验其是否存在。重复的校验和 bug 都是因为 `undefined` 值，这不仅浪费时间，还会因为过度的打字造成重复的压力。设定初始值会避免很多痛苦！

换言之，下面是一个反模式：

```
// Anti-pattern: Don't try this at home!
class Users extends React.Component {
  constructor(props) {
    super(props)
  }
  ...
  render() {
    return <div className="container">
      <h1>List of Users</h1>
    </div>
  }
}
```

不设置空的初始值



```

<table className="table-striped table-condensed table table-bordered
  > table-hover">
  <tbody>{(this.state.users && this.state.users.length>0) ?
    this.state.users.map((user)=>
      <tr key={user.id}>
        <td>{user.first_name} {user.last_name}</td>
        <td> {user.email}</td>
        <td> {user.ip_address}</td>
      </tr>) : ''}
    </tbody>
  </table>
</div>
}
}

```

校验是否存在  
(不需要初始值)

## 5.6 更新事件

如前所述，挂载事件通常被用来集成 React 和外部世界：其他框架、类库或数据存储。更新事件则和更新组件相关联。这些事件如下，从组件生命周期的开始到结束(参见仅包含更新生命周期事件的表 5.2 以及包含所有事件的表 5.1)：

- `componentWillReceiveProps(newProps)`
- `shouldComponentUpdate()`
- `componentWillUpdate()`
- `componentDidUpdate()`

表 5.2 组件更新时生命周期事件的调用

更新组件属性	更新组件状态	使用 <code>forceUpdate()</code> 更新
<code>componentWillReceiveProps()</code>	<code>shouldComponentUpdate()</code>	<code>componentWillUpdate()</code>
<code>shouldComponentUpdate()</code>	<code>componentWillUpdate()</code>	<code>render()</code>
<code>componentWillUpdate()</code>	<code>render()</code>	<code>componentDidUpdate()</code>
<code>render()</code>	<code>componentDidUpdate()</code>	
<code>componentDidUpdate()</code>		

### 5.6.1 `componentWillReceiveProps(newProps)`

当组件收到新的属性时会触发对 `componentWillReceiveProps(newProps)` 的调用。这个阶段被称为即将到来的属性转移。该事件允许在获取新属性和 `render()` 之前的这段时间里对组件进行拦截，以便添加一些逻辑。

`componentWillReceiveProps(newProps)` 方法接收新属性作为参数，它不会在组件初始化渲染时调用。如果想捕获新属性并在重新渲染之前相应地设置状态，该方法非常有用。`this.props` 对象指代的是旧的属性值。例如，以下代码片段设置 `opacity` 状态，它在 CSS 中取值为 0 或 1，依赖于布尔属性 `isVisible` 的取值(1=true, 0=false)：

```

componentWillReceiveProps(newProps) {
  this.setState({

```



```
    opacity: (newProps.isVisible) ? 1 : 0  
  })  
}
```

一般来说, 在 `componentWillReceiveProps(newProps)` 方法中调用 `setState()` 不会触发额外的重新渲染。

尽管接收了新的属性, 但这些属性不一定都有新的值(意味着与当前属性的值不同), 因为 React 无法知道属性值是否发生了变化。因此, 无论属性发生了什么变化, 只要有渲染(来自父组件或调用), `componentWillReceiveProps(newProps)` 每次都会被执行。因此, 不能假定 `newProps` 总是与当前属性不同。

与此同时, 重新渲染(调用 `render()`)并不一定意味着实际 DOM 的更新。是否更新以及在 DOM 中更新哪些内容的决定将委托给 `shouldComponentUpdate()` 和协调过程<sup>1</sup>。

### 5.6.2 shouldComponentUpdate()

接下来是 `shouldComponentUpdate()` 事件, 它在渲染之前调用。渲染以是否收到新属性和状态为先决条件。`shouldComponentUpdate()` 事件在初始渲染和调用 `forceUpdate()` 时不会被触发(参见表 5.1)。

可以通过实现 `shouldComponentUpdate()` 事件并 `return false` 来阻止 React 重新渲染。这在组件的属性和状态没有变化, 并且希望避免不必要的性能损耗(处理数百个组件)时非常有用。例如, 以下代码片段使用+二元运算符将布尔属性 `isVisible` 的值转换成一个数字, 并将其与 `opacity` 值进行比较:

```
shouldComponentUpdate(newProps, newState) {  
  return this.state.opacity !== + newProps.isVisible  
}
```

当 `isVisible` 为 `false` 时, `this.state.opacity` 为 0, 整个 `render()` 会被跳过; `componentWillUpdate()` 和 `componentDidUpdate()` 也不会被调用。本质上, 可以控制组件是否需要重新渲染。

### 5.6.3 componentWillUpdate()

`componentWillUpdate()` 事件会在渲染之前被调用, 并且先于接收到新的属性和状态。该方法在初始渲染时不会被调用。`componentWillUpdate()` 方法可以作为在更新发生之前执行准备工作的地方。应该避免在这里使用 `this.setState()`! 为什么? 你能想象在即将更新组件时又触发新的更新逻辑会发生什么吗? 听起来就是个坏主意。

如果 `shouldComponentUpdate()` 返回 `false`, `componentWillUpdate()` 将不会被调用。

---

<sup>1</sup> 关于 React 无法在 `componentWillReceiveProps(newProps)` 调用之前执行更智能检测的更多原因, 请阅读 React 团队成员 Jim Sproch 于 2016/01/08 撰写的文章 “(A => B)! => (B => A)” (<http://mng.bz/3WpG>)





### 5.6.4 componentDidMount()

`componentDidUpdate()` 事件在组件更新到 DOM 之后会立即触发。同样，该方法并不会在初始渲染时调用。`componentDidUpdate()` 在组件更新后撰写和 DOM 以及其他元素相关的代码时非常有用，因为在这个阶段，所有的更新都会被渲染到 DOM 中。

每当有组件挂载或更新时，都应该有对应的卸载方法。接下来的事件提供了一个可供放置卸载逻辑的地方。

## 5.7 卸载事件

在 React 中，卸载意味着从 DOM 中解绑或移除元素。这个事件类别中只有一个事件，它也是组件生命周期的最后一个类别。

### `componentWillUnmount()`

`componentWillUnmount()` 事件会在组件从 DOM 中卸载之前被调用。可以在该方法中添加任何必要的清理逻辑。例如，清除计时器、清理任何 DOM 元素或解除事件等这类在 `componentDidMount()` 中创建的逻辑。

## 5.8 一个简单示例

假设你的任务是创建一个笔记本 Web APP(以在线保存文本)。组件已经实现，但来自用户的初始反馈是：如果他们无意中关闭了窗口(或选项卡)，就会丢失内容。下面我们来一起实现图 5.6 所示的确认对话框。

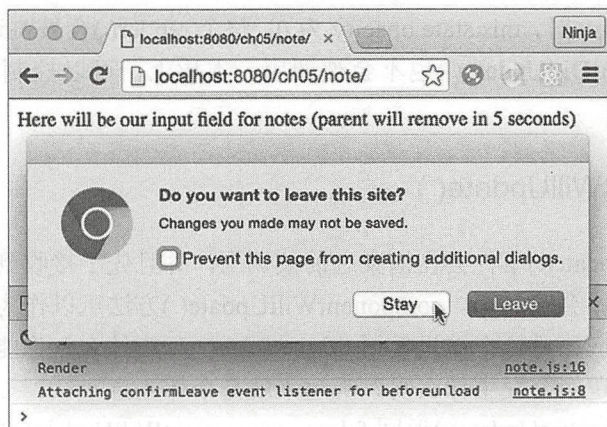


图 5.6 当用户试图离开页面时的确认对话框

要实现这样的对话框，需要监听特殊的 `window` 事件。棘手的部分是：需要在元素无用之时对事件进行清理，因为如果元素被删除，但事件仍存在，将导致内存泄漏！解决这个问题的最好方法是在挂载时绑定事件，在卸载时移除事件。

项目结构如下：



```

/note
  /jsx
    note.jsx
    script.jsx
  /js
    note.jsx
    react.js
    react-dom.js
    script.js
index.html

```

`window.onbeforeunload` 这个浏览器原生事件(需要附加代码以实现跨浏览器支持)非常简单:

```

window.addEventListener('beforeunload',function() {
  let confirmationMessage = 'Do you really want to close?'
  e.returnValue = confirmationMessage // Gecko, Trident, Chrome 34+
  return confirmationMessage         // Gecko, WebKit, Chrome < 34
})

```

以下方式同样可以工作:

```

window.onbeforeunload = function() {
  ...
  return confirmationMessage
}

```

将上述代码放入 `componentDidMount()` 方法的事件监听器中, 并在 `componentWillUnmount()` 中移除该事件监听器(ch05/note/jsx/note.jsx), 如代码清单 5.4 所示。

#### 代码清单5.4 添加和移除事件监听器

```

class Note extends React.Component {
  confirmLeave(e) {
    let confirmationMessage = 'Do you really want to close?'
    e.returnValue = confirmationMessage // Gecko, Trident, Chrome 34+
    return confirmationMessage         // Gecko, WebKit, Chrome <34
  }
  componentDidMount() {
    console.log('Attaching confirmLeave event listener for beforeunload')
    window.addEventListener('beforeunload', this.confirmLeave)
  }
  componentWillUnmount() {
    console.log('Removing confirmLeave event listener for beforeunload')
    window.removeEventListener('beforeunload', this.confirmLeave)
  }
  render() {
    console.log('Render')
    return Here will be our input field for notes (parent will remove in
    {this.props.secondsLeft} seconds)
  }
}

```



在 Note 元素被移除时,想检查一下代码是如何工作的吗? 出于这个原因,需要删除 Note 元素,以便它可以被卸载。因此,下一步是实现可以添加和删除 Note 元素的父组件。我们使用计时器(setInterval())来实现,如代码清单 5.5(ch05/note/jsx/script.jsx)和图 5.7 所示。

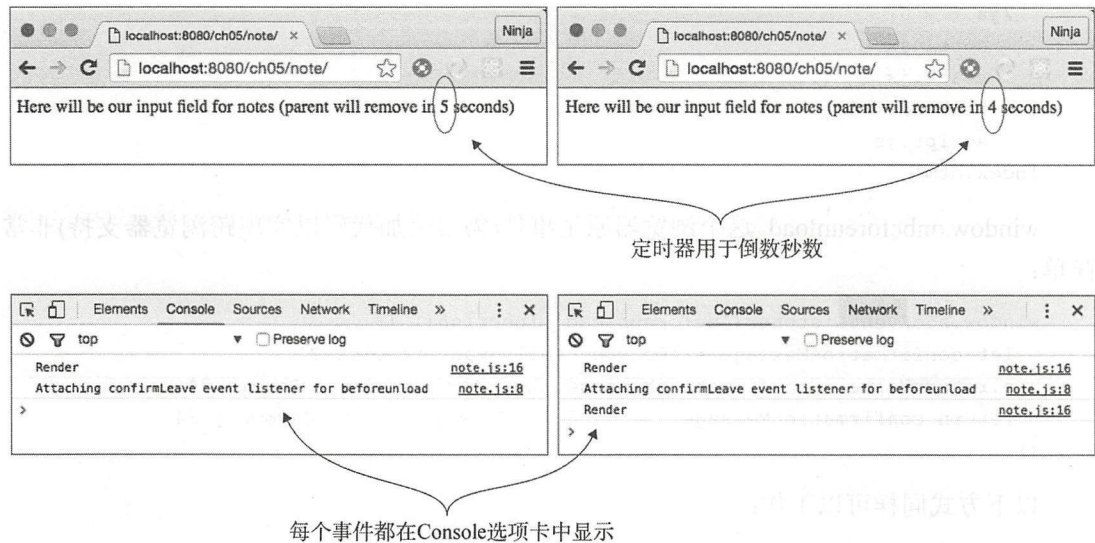


图 5.7 在 5、4、...秒时, Note 元素将会被另一个元素替换

代码清单5.5 在移除之前渲染Note元素

```
let secondsLeft = 5

let interval = setInterval(()=>{
  if (secondsLeft == 0) {
    ReactDOM.render(
      <div>
        Note was removed after {secondsLeft} seconds.
      </div>,
      document.getElementById('content')
    )
    clearInterval(interval)
  } else {
    ReactDOM.render(
      <div>
        <Note secondsLeft={secondsLeft}/>
      </div>,
      document.getElementById('content')
    )
  }
  secondsLeft--
}, 1000)
```

图 5.8 显示了结果(带控制台日志): 渲染、绑定事件监听器、渲染多于 4 次、移除事件监听器。如果不删除 componentWillUnmount()中的事件监听器(可以通过注释掉该方法来查看),页面仍然会有让人讨厌的对话框出现,即使 Note 元素早已经消失,如图 5.9 所示。





这不是好的交互体验，并可能会导致 bug。可以使用这个生命周期事件来清理组件。

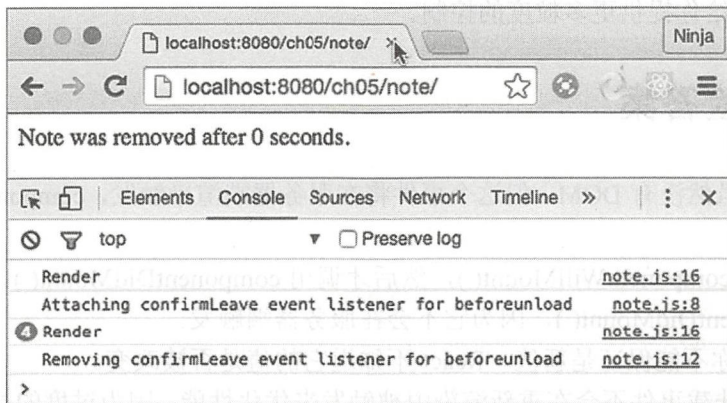


图 5.8 Note 元素被 div 替换，并且在用户离开页面时也不会弹出确认对话框

React 团队正在倾听来自开发者的反馈。大多数生命周期事件允许开发者调整组件的行为。可以把生命周期事件想象成黑腰带女忍者拥有的绝技。可以在没有它们的情况下编码，但是使用它们后，你的代码会更加强大。有趣的是，仍然有关于最佳实践和使用的讨论。React 还在持续发展，未来的生命周期事件可能会发生变化或增加。如果需要查阅官方文档，参见 <https://facebook.github.io/react/docs/react-component.html>。

## 5.9 测验

1. `componentWillMount()` 会在服务器端调用。这么理解正确还是错误？
2. 哪个事件将首先触发，`componentWillMount()` 还是 `componentDidMount()`？
3. 下面哪个事件是放置 AJAX 请求以从服务器端为组件获取数据的好地方？`componentWillUnmount()`、`componentHasMounted()`、`componentDidMount()`、`componentWillReceiveProps()` 还是 `componentWillMount()`？
4. `componentWillReceiveProps()` 方法意味着当前元素将会有一次重新渲染(来自父结构)，并且你明确知道新的属性值。这么理解正确还是错误？
5. 挂载事件会在每一次重新渲染时触发多次。这么理解正确还是错误？

## 5.10 小结

- `componentWillMount()` 会在客户端和服务端调用，`componentDidMount()` 只会在客户端调用。
- 挂载事件通常被用来整合 React 和其他库，并从服务器或存储中获取数据。
- 使用 `shouldComponentUpdate()` 来优化渲染。
- 使用 `componentWillReceiveProps()` 以借助新的属性来执行状态改变。
- 卸载事件通常用于清理。

- 更新事件提供了一个地方来存放依赖新属性或状态的逻辑，并且它们将在更新视图时，给你提供更多粒度的控制。

## 5.11 测验答案

1. 正确。虽然没有 DOM，但这个事件将在服务器端渲染触发，`componentDidMount()` 不会。
2. 先调用 `componentWillMount()`，然后才调用 `componentDidMount()`。
3. `componentDidMount()`，因为它不会在服务器端触发。
4. 错误。你不能保证是新值。React 不知道它的值是否被改变。
5. 错误。挂载事件不会在重新渲染中被触发来优化性能，因为过度的挂载是一项相对昂贵的操作。

# 第 6 章

## React 事件处理

本章内容:

- 在 React 中处理 DOM 事件
- 响应 React 不支持的 DOM 事件
- 将 React 和其他类库集成: jQuery UI 事件

迄今为止,我们已经了解了如何渲染无交互的 UI 视图。换言之,这只是在展现数据。例如,你已经构建了一个不接收用户输入(例如设置时区)的时钟程序。

多数时候,需要构建足够智能的元素来响应用户的操作,而不仅仅是静态的 UI。如何响应用户的操作呢?例如单击或拖曳鼠标。

本章提供如何在 React 中处理事件的解决方案。然后,在第 7 章,将会应用这些知识来处理 Web 表单及其元素。我提到过,React 只支持某些事件,在本章中,将会向你展示如何处理 React 不支持的那些事件。

### 6.1 在 React 中处理 DOM 事件

下面来看看如何通过为这些操作定义事件处理程序来让 React 元素响应用户操作。方法是定义事件处理程序(函数定义)并在 JSX 中将它作为元素的属性值,或作为纯 JavaScript 中元素的属性值(当使用 `createElement()` 而不是 JSX 时)。对于事件名称属性,使用标准的 W3C DOM 事件名称,以驼峰规范命名,例如 `onClick` 或 `onMouseOver`:

```
onClick={function() {...}}
```

或者

```
onClick={() => {...}}
```



例如，在 React 中，可以定义一个事件监听器，并且在用户单击按钮时触发。在事件监听器中，打印出 `this` 上下文。`event` 对象是本地 DOM 事件对象的增强版本(称为 `Synthetic-Event`):

```
<button onClick={(function(event) {
  console.log(this, event)
}).bind(this)}>
  Save
</button>
```

`bind()` 是必需的，以便在事件处理程序中引用类(React 元素)的实例。如果不绑定，`this` 会是 `null` 值(严格模式下)。在以下几种场景中，不应该使用 `bind(this)` 将上下文绑定到类的实例上：

- 不需要通过 `this` 来引用类的实例时。
- 使用旧的风格 `React.createClass()` 而不是 ES6+ 的类风格时，因为 `createClass()` 会为你自动绑定。
- 使用箭头函数时 `(( )=>{ })`。

还可以使用类方法作为 `onClick` 事件的处理程序(将其命名为 `handleSave()`)来让代码看起来更加整洁。想象一下 `SaveButton` 组件，当单击时，打印 `this` 和 `event` 的值，但如图 6.1 和代码清单 6.1(ch06/button/jsx/button.jsx)所示，使用一个类方法来实现。

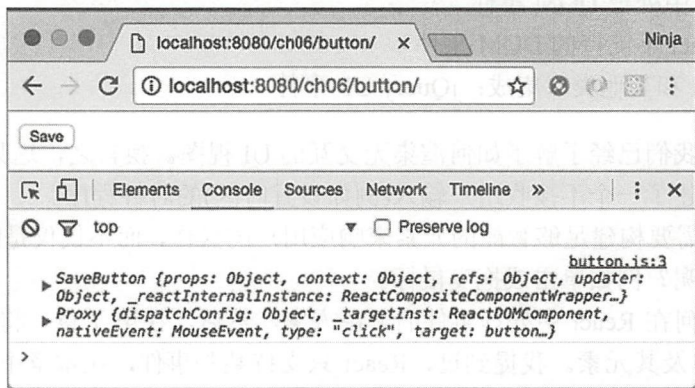


图 6.1 单击 Save 按钮将打印 `this` 的值: `SaveButton`

#### 代码清单6.1 定义一个作为类方法的事件处理程序

```
class SaveButton extends React.Component {
  handleSave(event) {
    console.log(this, event)
  }
  render() {
    return <button onClick={this.handleSave.bind(this)}>
      Save
    </button>
  }
}
```

为 `onClick` 事件传递通过 `bind()` 返回的函数定义

这就是 `Save` 按钮将要打印的 `this` 和 `event` 日志。

此外，可以在类的 `constructor()` 中为事件处理程序执行绑定。从本质上说，这没有区别，但如果不止一次在 `render()` 中使用相同的方法，`constructor()` 绑定会减少重复绑定。下面是相同的按钮，但使用了 `constructor()` 绑定：

```
class SaveButton extends React.Component {
  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)
  }
  handleClick(event) {
    console.log(this, event)
  }
  render() {
    return <button onClick={this.handleClick}>
      Save
    </button>
  }
}
```

绑定this上下文到类的实例上，以便事件处理程序中的this可以指向类的实例

为onClick设置函数定义

绑定事件处理程序是我最喜欢和推荐的做法，因为它消除了复制并将所有绑定整齐地放置在一起。

表 6.1 列出了 React v15 支持的事件类型。注意在事件名称中使用驼峰命名规范，与 React 中的其他属性名称一致。

表 6.1 React v15 支持的 DOM 事件

事件组	React 支持的事件
鼠标事件	onClick、onContextMenu、onDoubleClick、onDrag、onDragEnd、onDragEnter、onDragExit、onDragLeave、onDragOver、onDragStart、onDrop、onMouseDown、onMouseEnter、onMouseLeave、onMouseMove、onMouseOut、onMouseOver、onMouseUp
键盘事件	onKeyDown、onKeyPress、onKeyUp
剪贴板事件	onCopy、onCut、onPaste
表单事件	onChange、onInput、onSubmit
焦点事件	onFocus、onBlur
触摸事件	onTouchCancel、onTouchEnd、onTouchMove、onTouchStart
UI 事件	onScroll
滚轮事件	onWheel
选择事件	onSelect
图片事件	onLoad、onError
动画事件	onAnimationStart、onAnimationEnd、onAnimationIteration
过渡事件	onTransitionEnd

如你所见，React 支持多种类型的标准事件。如果对比 <https://developer.mozilla.org/en-US/docs/Web/> 上的标准事件列表，就会发现 React 对事件的支持非常广泛，并且可以肯定 React 团队将会在未来支持更多的事件。更多相关的信息和事件名称，请访问 <http://facebook.github.io/react/docs/events.html> 上的文档页。

### 6.1.1 捕获和冒泡阶段

如前所述，React 是声明式而不是命令式的，它消除了对操作对象的需求，而且不会像使用 jQuery 那样在代码中绑定事件(例如: `$('#btn').click(handleSave)`)。相反，在 JSX 中，事件被声明为属性(例如，`onClick={handleSave}`)。如果要声明鼠标事件，属性名可以是表 6.1 中受支持的任何事件。属性值就是事件处理程序。

例如，如果要定义鼠标悬停事件，可以使用 `onMouseOver`，如下所示。当把鼠标移到 `<div>` 的红色边框内悬停时，会在开发者工具或 Firebase 控制台显示 “mouse is over”。

```
<div
  style={{border: '1px solid red'}}
  onMouseOver={()=>{console.log('mouse is over')}} >
  Open DevTools and move your mouse cursor over here
</div>
```

之前展示的事件，例如 `onMouseOver`，是由冒泡阶段的事件触发的。如你所知，在冒泡和目标阶段之前，还有捕获阶段。捕获阶段最先，从 window 向下到目标元素；然后是目标阶段；当事件沿着节点树返回到 window 时，就是冒泡阶段，如图 6.2 所示。

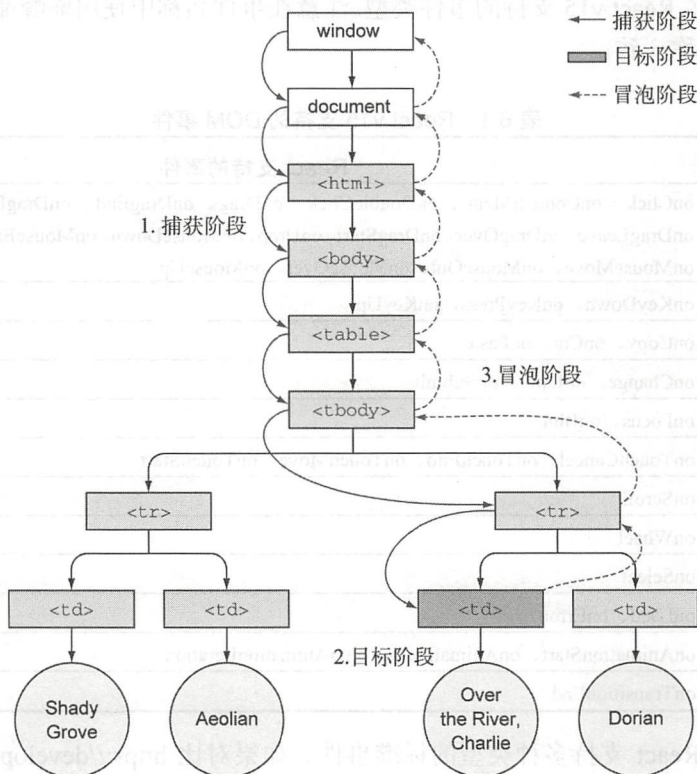


图 6.2 捕获、目标和冒泡阶段

当元素及其祖先元素上有相同的事件时，阶段区分就显得更重要了。在冒泡模式中，事件首先被最里面的元素(目标)捕获并处理，然后传播到外部元素(祖先，从目标的父元素开始)。在捕获模式下，事件首先被最外层元素捕获，然后传播到内部元素。



要为捕获阶段注册事件监听器，请将 `Capture` 添加到事件名称中。例如，可以使用 `onMouseOverCapture` 而不是 `onMouseOver` 来处理捕获阶段的 `mouseover` 事件。这对表 6.1 中的所有事件名称都适用。

为了论证这一点，可以假设一个带常规(冒泡)事件的 `<div>` 和一个捕获事件。这些事件分别用 `onMouseOver` 和 `onMouseOverCapture` 定义(ch06/mouse-capture/jsx/mouse.jsx)，如代码清单 6.2 所示。

代码清单6.2 捕获事件的后面紧接着是冒泡事件

```
class Mouse extends React.Component {
  render() {
    return <div>
      <div
        style={{border: '1px solid red'}}
        onMouseOverCapture={() => {
          console.log('mouse over on capture event')
          console.dir(event, this)}}.bind(this)
        onMouseOver={() => {
          console.log('mouse over on bubbling event')
          console.dir(event, this)}}.bind(this)} >
        Open DevTools and move your mouse cursor over here
      </div>
    </div>
  }
}
```

`div` 容器有一条 1 像素的红色边框，包含一些文本，如图 6.3 所示。因此，你可以知道光标应该悬停在哪里。每个 `mouseover` 事件将记录事件类型以及事件对象(因为使用了 `console.dir()`，对象结果隐藏在图 6.3 所示开发者工具的 Proxy 节点之下)。

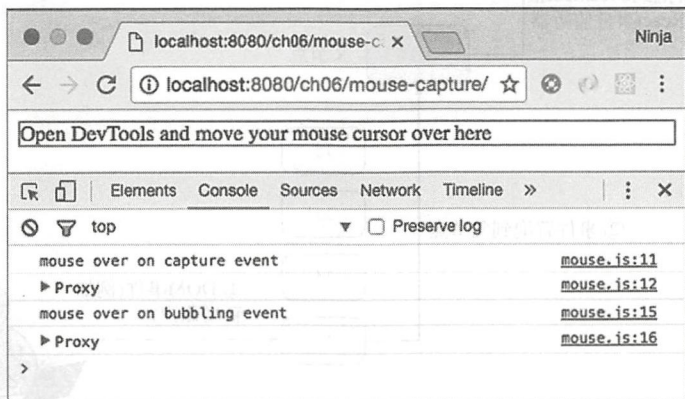


图 6.3 捕获事件发生在常规事件之前

毫不惊奇的是，捕获事件将首先被记录下来。可以使用此行为来停止事件的传播，并在事件之间设置优先级。

了解如何对事件进行响应很重要，因为事件是 UI 的基石。第 7 章将深入研究 React 事件。

## 6.1.2 React 事件的内幕

React 中的事件与 jQuery 或纯 JavaScript 中事件的工作方式有所不同,后者通常将事件监听器直接附加到 DOM 节点上。这种情况下,在 UI 生命周期内新增和移除事件可能会出现问題。例如,假设有一个账户列表,每个账户都可以删除或编辑,新账户也可以添加到列表中。HTML 大致如下,每个账户元素<li>由 id 唯一标识:

```
<ul id="account-list">
  <li id="account-1">Account #1</li>
  <li id="account-2">Account #2</li>
  <li id="account-3">Account #3</li>
  <li id="account-4">Account #4</li>
  <li id="account-5">Account #5</li>
  <li id="account-6">Account #6</li>
</ul>
```

如果删除和新增账户的操作十分频繁,管理事件会变得困难。更好的方法是在父节点(account-list)上添加事件监听器,监听冒泡事件(如果事件没有在较低级别上被捕获,则会冒泡到 DOM 树的上层节点上)。在内部,React 通过一张映射表来跟踪高层级元素和目标元素之间的事件绑定。这将允许 React 从根元素(document)开始跟踪目标,如图 6.4 所示。

下面来看看代码清单 6.2 所示的关于 Mouse 组件的例子中,委托到父元素的事件是如何起作用的。有一个绑定了 React mouseover 事件的<div>元素,我们想要检查这个元素上的事件。

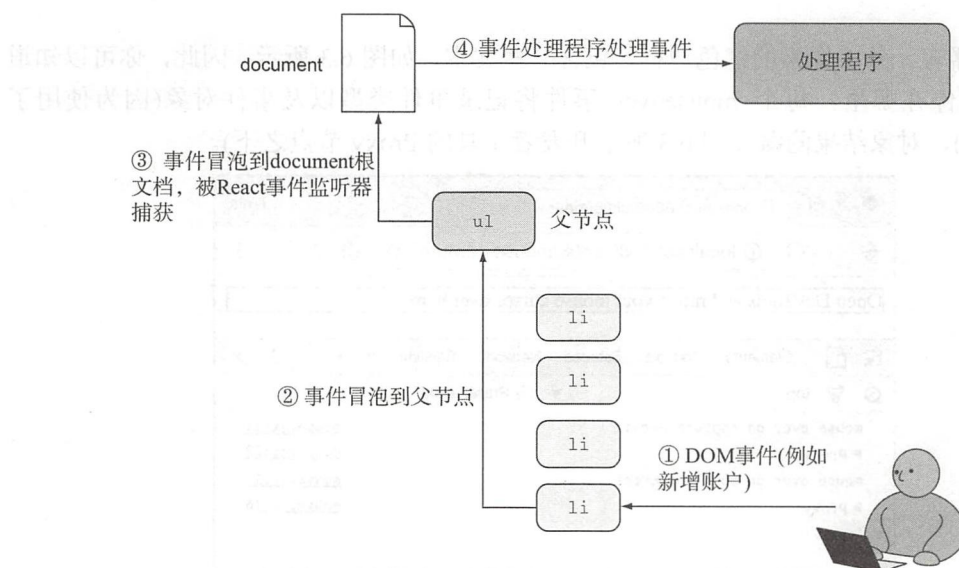


图 6.4 DOM 事件(①)冒泡到它的父节点(②和③),被一个常规的(冒泡阶段)React 事件监听器(④)捕获,因为在 React 中,事件会在根节点中被捕获

如果打开 Chrome 开发者工具或 Firefox 工具,在 Elements 或 Inspector 标签页(或使用 Chrome 右键菜单中的 Inspector 选项,或使用 Firefox 右键菜单中的 Inspector Element 选项)中选择 data-reactroot 元素,然后就可以在 Console(开发者工具/Firefox 中的另一个标签页)

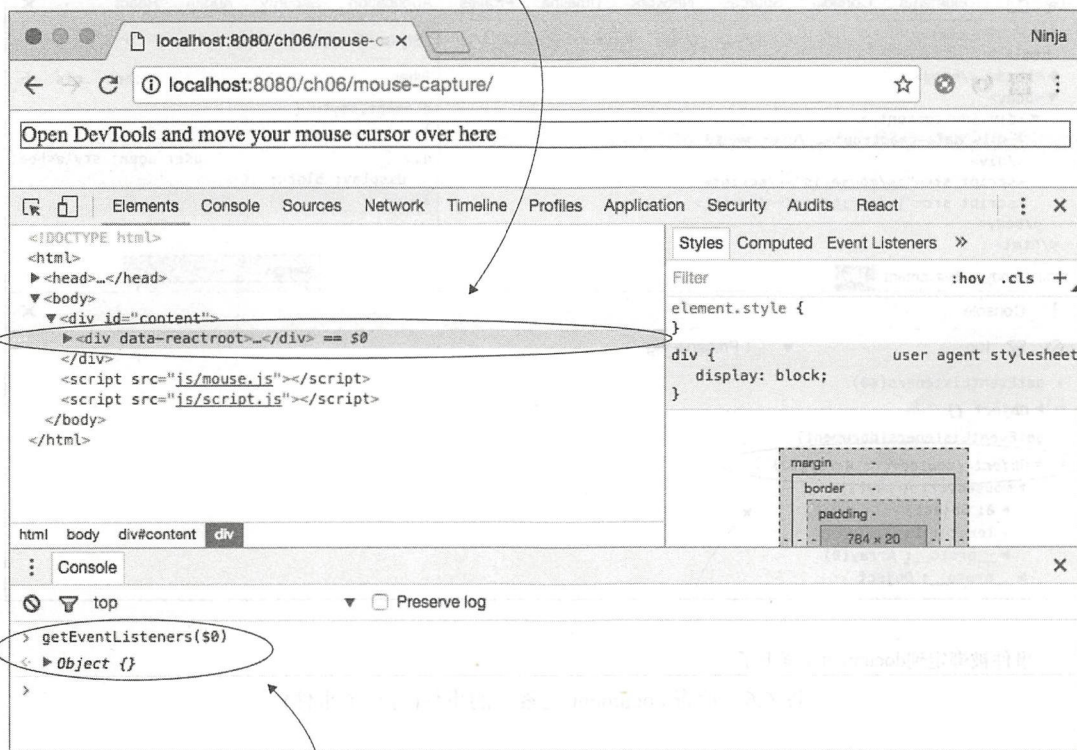
中输入\$0 并按回车键来引用这个元素。这是一个很棒的小技巧。

有趣的是, 这个 DOM 节点<div>没有任何事件监听器。\$0 既是<div>, 也是 reactroot 元素, 如图 6.5 所示。因此, 可以使用开发者工具 Console 中的全局方法 `getEventListeners()` 来检查这个特定元素(DOM 节点)上绑定了哪些事件:

```
getEventListeners($0)
```

结果是一个空对象 `{}`。React 并未绑定事件监听器到这个 reactroot 节点<div>上。将鼠标悬停在元素上会打印出日志, 可以清楚地看到事件正在被捕获! 那么它去哪儿了呢?

① 在Elements标签页中选择data-reactroot



② 输入\$0, 并按回车键

图 6.5 检查<div>元素上的事件(一无所获)

随意使用 `<div id="content">` 或红色边框的 `<div>` 元素重复这个过程。对于 Elements 标签页中每个当前选定的元素, `$0` 会指向它们。因此, 选择一个新元素, 重复 `getEventListeners($0)`。依然一无所获, 是吗?

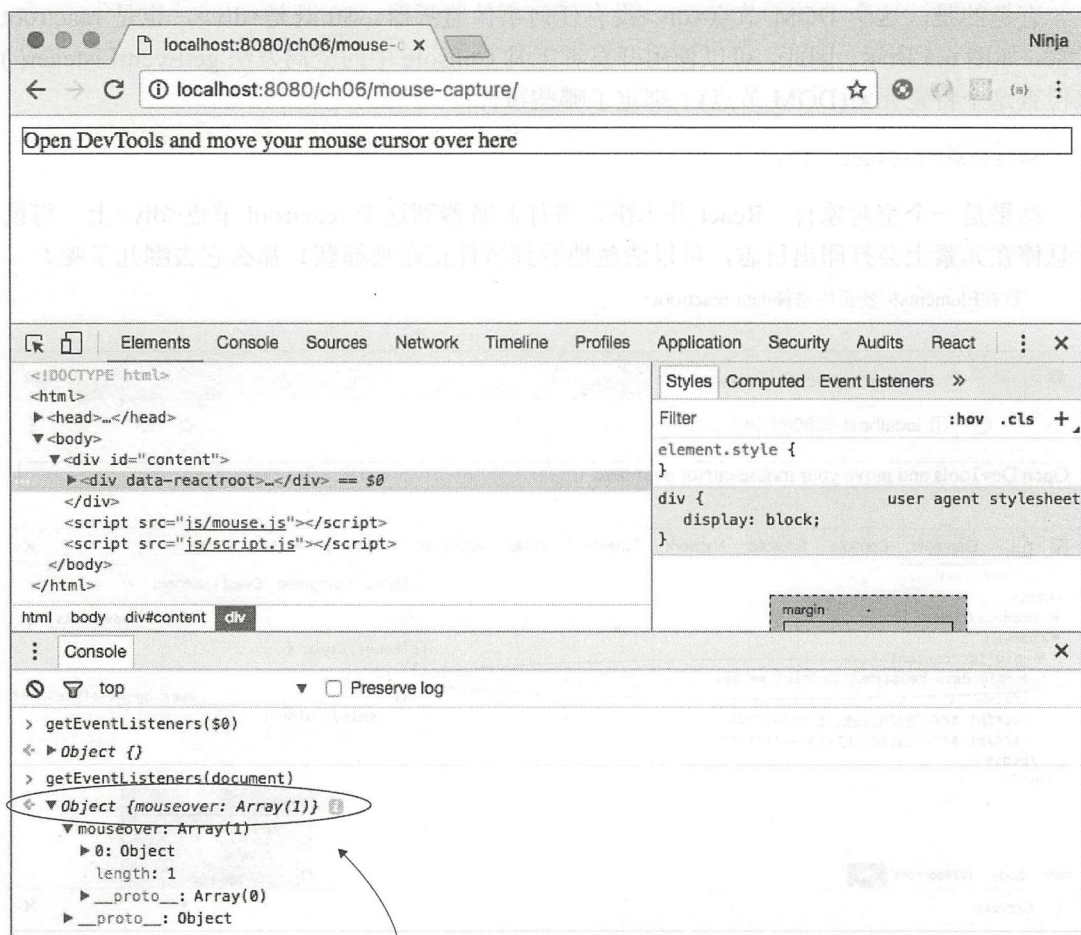
好了。我们通过在控制台执行以下代码来检查 document 上的事件:

```
getEventListeners(document)
```

终于有事件了: `Object{mouseover:Array{1}}`, 如图 6.6 所示。现在你知道了, React 是把事件监听器绑定到顶级父节点, 也就是所有元素的祖先节点——document 元素上了。



事件并没有绑定到诸如<div>或拥有 data-reactroot 属性的元素这样的单个节点上。



事件被绑定到document元素上了

图 6.6 检查 document 元素上的事件(有一个事件)

接下来，可以通过在控制台中调用下面这行代码来移除事件：

```
getEventListeners(document).mouseover[0].remove()
```

现在，当移动光标时，“mouse is over”这条消息不会再出现了。绑定到 document 上的事件监听器消失了，说明 React 将事件绑定到 document 上了，而不是绑定到每个元素。这可以使 React 更快，尤其在处理列表时。这与 jQuery 的工作方式相反：jQuery 中的事件被绑定到单个元素上。由衷敬佩 React 在性能上的表现。

如果有其他元素拥有相同类型的事件(例如，两个 mouseover)，将它们绑定到一个事件上并通过 React 内部映射到正确的目标元素，如图 6.7 所示。谈到目标元素，可以从事件对象中获取目标节点(事件发生的位置)的信息。

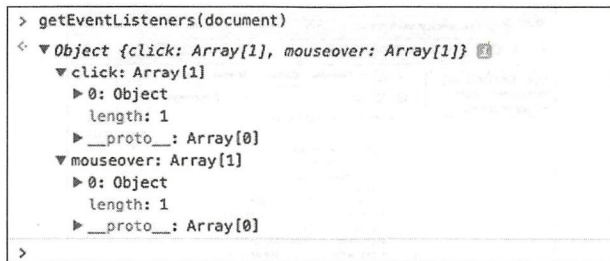


图 6.7 React 在根文档上复用事件监听器，因此当有多个元素上有 mouseover 事件时，你能看到每种事件类型只有一个事件

### 6.1.3 使用 React SyntheticEvent 事件对象

浏览器可以在对 W3C 规范的实现上有所不同(请参阅 [www.w3.org/TR/DOM-Level-3-Events](http://www.w3.org/TR/DOM-Level-3-Events))。使用 DOM 事件时，传递给事件处理程序的事件对象可能具有不同的属性和方法。当编写事件处理程序时，这可能会导致跨浏览器兼容问题。例如，要获取 IE8 中的目标元素，需要访问 event.srcElement；而在 Chrome、Safari 和 Firefox 中，则需要使用 event.target：

```
var target = event.target || event.srcElement
console.log(target.value)
```

当然，2018 年的跨浏览器兼容问题相比 2006 年要好很多。但是由于浏览器实现之间模糊的差异，你会想要花时间来阅读规范和调试问题吗？我不会。

跨浏览器兼容问题很糟糕，因为用户应该在不同的浏览器中拥有相同的体验。通常，需要添加额外的代码(如 if/else 语句)来解释浏览器 API 的差异。还必须在不同的浏览器中执行更多的测试。简而言之，解决跨浏览器兼容问题要比 CSS 问题、IE8 问题或佩戴时髦眼镜、认真严谨的设计人员让人更加烦恼。

React 提出了一个解决方案：封装浏览器的原生事件。这确保了事件与 W3C 规范一致，而无论页面运行在何种浏览器上。React 在底层使用自己的特殊类(SyntheticEvent)来定义合成事件。SyntheticEvent 类的实例会被传递给事件处理程序。例如，要访问一个合成事件对象，可以将参数 event 添加到事件处理程序中，如代码清单 6.3(ch06/mouse/jsx/mouse.jsx)所示。这样，便会在控制台中输出事件对象，如图 6.8 所示。

代码清单6.3 事件处理程序接收一个合成事件

```
class Mouse extends React.Component {
  render() {
    return <div>
      <div
        style={{border: '1px solid red'}}
        onMouseOver={((event)=>{
          console.log('mouse is over with event')
          console.dir(event)}}} >
        Open DevTools and move your mouse cursor over here
      </div>
    </div>
  }
}
```

定义一个 event 参数

访问 SyntheticEvent 对象并以可交互的方式(dir)输出



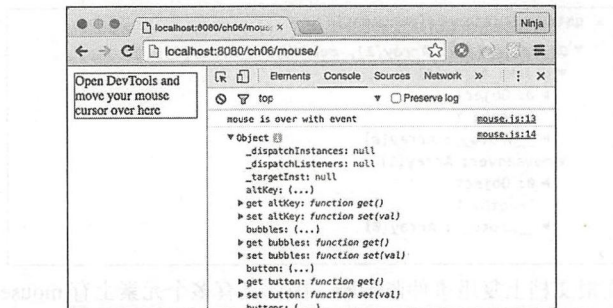


图 6.8 将鼠标悬停在方框上，会在开发者工具的控制台中打印事件对象

如前所述，可以将事件处理程序移到组件方法或独立的函数中。例如，可以使用 ES6+/ES2015+ 的类方法语法创建 `handleMouseOver()` 方法，并通过在 `render()` 方法的返回值中使用 `{this.handleMouseOver.bind(this)}` 来引用。`bind()` 方法是必需的，用来为事件处理程序传递适当的 `this` 值。当使用上一例中的箭头函数语法时，这个过程会自动完成。同样，在 `createClass()` 语法中也会自动完成。但是 `class` 语法不行。当然，如果不在方法中使用 `this`，就不用绑定它，用 `onMouseOver={this.handleMouseOver}` 就可以了，如代码清单 6.4 所示。

名称 `handleMouseOver()` 是随意取的(不像第 5 章中的生命周期事件名称)，只要你和团队了解它，就不必遵循任何约定。大多数情况下，会给事件处理程序加上名为 `handle` 的前缀，以便将其与常规的方法区分开来，并且会包括事件名称(例如 `mouseover`)或操作名称(例如 `save`)。

#### 代码清单 6.4 事件处理程序作为类方法：在 `render()` 中绑定

```
class Mouse extends React.Component {
  handleMouseOver(event) {
    console.log('mouse is over with event')
    console.dir(event.target)
  }
  render() {
    return <div>
      <div
        style={{border: '1px solid red'}}
        onMouseOver={this.handleMouseOver.bind(this)} >
        Open DevTools and move your mouse cursor over here
      </div>
    </div>
  }
}
```

React 的事件与大多数原生浏览器事件具有相同的属性和方法，例如 `stopPropagation()` / `preventDefault()`、`target` 和 `currentTarget`。如果找不到原生事件的属性或方法，可以通过 `nativeEvent` 访问浏览器原生事件对象：

```
event.nativeEvent
```

下面是 React v15 中 `SyntheticEvent` 接口的一些属性和方法：



- `currentTarget`: `DOMEventTarget`, 捕获事件(可以是目标事件或目标的父事件)的元素。
- `target`: `DOMEventTarget`, 触发事件的元素。
- `nativeEvent`: `DOMEvent`, 浏览器原生事件对象。
- `preventDefault()`: 阻止浏览器的默认行为, 例如链接或表单提交按钮。
- `isDefaultPrevented()`: 布尔值。如果默认行为被阻止, 则返回 `true`。
- `stopPropagation()`: 阻止事件的传播。
- `isPropagationStopped()`: 布尔值。如果事件传播被阻止, 则返回 `true`。
- `type`: 标签名字字符串。
- `persist()`: 从事件池中删除合成事件, 并允许用户代码保留其引用。
- `IsPersistence()`: 布尔值。如果 `SyntheticEvent` 被移除事件池, 则返回 `true`。

上述事件对象的 `target` 属性持有触发该事件元素的 DOM 节点的引用, 而不是捕获该事件元素的 DOM 节点(`currentTarget`), 参见 <https://developer.mozilla.org/en-US/docs/Web/API/Event/target>。大多数情况下, 构建 UI 时, 除了捕获之外, 还需要获取输入字段中的文本, 可以从 `event.target.value` 中获取。

一旦事件处理程序执行结束, 合成事件将被取消(意味着不可用)。因此, 可以使用一个变量来保存事件的引用, 以便稍后访问或(未来)在回调函数中异步地访问。例如, 可以将事件对象保存在全局变量 `e` 中, 如代码清单 6.5 所示(ch06/mouseevent/jsx/mouse.jsx)。

代码清单6.5 取消合成事件

```
class Mouse extends React.Component {
  handleMouseOver(event) {
    console.log('mouse is over with event')
    window.e = event // Anti-pattern
    console.dir(event.target)
    setTimeout(()=>{
      console.table(event.target)
      console.table(window.e.target)
    }, 2345)
  }
  render() {
    return <div>
      <div
        style={{border: '1px solid red'}}
        onMouseOver={this.handleMouseOver.bind(this)}>
        Open DevTools and move your mouse cursor over here
      </div>
    </div>
  }
}
```

在方法中使用event对象及其属性

默认情况下, 无法在异步回调中使用事件或通过window.e来访问

这样做会得到警告: 出于性能原因, React 会重用合成事件(参见图 6.9):

```
This synthetic event is reused for performance reasons. If you're seeing this,
you're accessing the property `target` on a released/nullified synthetic
event. This is set to null.
```

如果在事件处理程序结束后需要保留合成事件, 请使用 `event.persist()` 方法。使用它, 事件对象将不会被重用和取消。

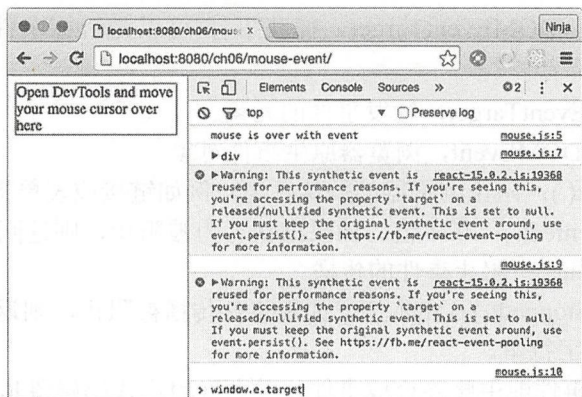


图 6.9 保存合成事件以便后续使用默认是不可行的，会产生警告

我们已经看到，React 会绑定浏览器原生事件，这意味着 React 将会对浏览器原生事件对象进行跨浏览器的封装。这样做的好处是：事件在几乎所有浏览器中都是一致的。大多数情况下，React 事件拥有几乎所有原生事件方法，包括 `event.stopPropagation()` 和 `event.preventDefault()`。但是，如果仍然需要访问原生事件，可以通过 `event.nativeEvent` 属性访问。显然，如果直接使用原生事件，则需要了解和处理遇到的任何浏览器间的差异。

### 6.1.4 使用事件和状态

把状态和事件一起使用，或者换句话说，能在响应事件时改变组件的状态，会给你带来响应用户操作的交互式 UI 体验。这将会非常有趣，因为可以捕获任何事件并根据这些事件和应用逻辑来修改视图。这将使你的组件独立，因为它们不需要任何外部代码和展示。

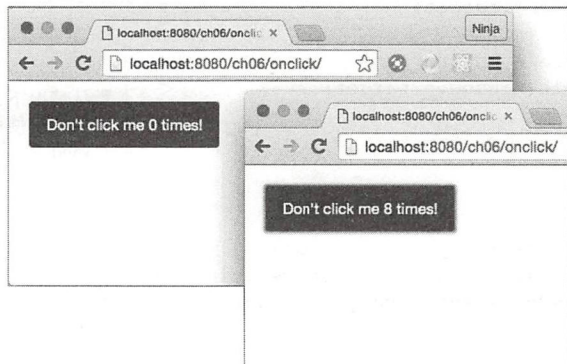


图 6.10 单击按钮将增加计数器计数，初始值为 0

例如，我们来实现一个带有计数器标签的按钮，计数器的值从 0 开始，如图 6.10 所示。每单击一次按钮，按钮上显示的数字就会加 1(1、2、3 等)。

从实现以下方法开始：

- `constructor()`: `this.state` 等于 1，在视图可用之前需要将 `counter` 设置为 0。
- `handleClick()`: 增加计数的事件处理程序。
- `render()`: 返回按钮 JSX 的渲染()方法。

`click()` 方法与 React 组件的其他方法没有什么不同。还记得第 3 章的 `getUrl()` 和本章



前面的 `handleMouseOver()` 吗？`click()` 的声明和它们类似，除了必须手动绑定 `this` 上下文以外。`handleClick()` 方法将 `counter` 状态设置为当前 `counter` 值加 1，如代码清单 6.6 所示 (ch06/onclick/jsx/content.jsx)。

代码清单6.6 单击以更新状态

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.state = {counter: 0}
  }
  handleClick(event) {
    this.setState({counter: ++this.state.counter})
  }
  render() {
    return (
      <div>
        <button
          onClick={this.handleClick.bind(this)}
          className="btn btn-primary">
          Don't click me {this.state.counter} times!
        </button>
      </div>
    )
  }
}
```

设置初始状态  
counter为0

将counter值加1

为onClick事件绑定事件  
监听器handleClick

显示  
counter  
状态值

### 调用和定义

提醒一下：你是否注意到 `this.handleClick()` 虽然是代码清单 6.6 中的一个方法，但当它被分配给 `onClick` (也就是 `<button onClick={this.handleClick}>`) 时，并不会被调用？换言之，花括号内 `this.handleClick` 的后面并没有括号 `(( ))`。那是因为这里需要的是传递一个函数的定义，而不是调用它。函数是 JavaScript 中的一等公民，在本例中，将函数定义作为值传递给 `onClick` 属性。

另一方面，调用 `bind()` 方法可以确保使用适当的 `this` 值，但 `bind()` 方法返回的是一个函数的定义。所以，`onClick` 的值仍然是一个函数的定义。

请记住，如前所述，`onClick` 不是真正的 HTML 属性，但在语法上看起来就像任何其他 JSX 声明 (例如，`className={btnClassName}` 或 `href={this.props.url}`)。

当单击按钮时，会看到每单击一次计数器就会加 1。图 6.10 显示单击按钮 8 次后，`counter` 的值现在是 8，但初始值是 0。很棒，不是吗？

和 `onClick` 和 `onMouseOver` 类似，可以使用 React 支持的任何 DOM 事件。本质上，你定义了视图和一个改变状态的事件处理程序。不必强制修改视图展现。这是声明式风格的强大之处。

下一节将学习如何给予元素传递事件处理程序和其他对象。

### 6.1.5 传递事件处理程序和属性

考虑这样一种情况：有一个无状态的按钮组件，组件上只有样式。如何为它添加事件监听器以便它可以触发一些行为呢？



我们先回头看看属性。属性是不可变的，这意味着它们不能被改变。它们通过父组件传递给子组件。因为函数是 JavaScript 的一等公民，所以可以把函数当成属性传递给子元素并把它作为事件处理程序。

前面问题(触发无状态组件的事件)的解决方案是将事件处理程序作为属性传递给这个无状态组件，并在其中使用(调用这个函数)这个属性(事件处理程序)。例如，我们将前一个示例的功能分解为两个组件：ClickCounterButton 和 Content。第一个是无状态的(木偶)组件，第二个是有状态的(智能)组件。

#### 展示/木偶组件和容器/智能组件

木偶和智能组件有时分别称为展示组件和容器组件，这种二分法与无状态和有状态有关，但也并非总是这样。

多数情况下，展示组件没有状态，可以是无状态组件或函数组件。但并非总是这样，因为可能需要有一些与展示相关的状态。

展示/木偶组件经常使用 `this.props.children` 并渲染 DOM 元素。另一方面，容器/智能组件描述事物如何在没有 DOM 元素的情况下工作，有状态，通常会使用高阶组件模式，并连接到数据源。

使用木偶组件和智能组件的组合是最佳实践。这样做可以使事情保持简单，使关注点得到更好的分离。

当运行代码时，`counter` 的值会随着每次单击而增加。从前面的例子中看，按钮和计数器在视觉上没有什么变化(参见图 6.10)。但在内部，除了仍然持有所有逻辑的 Content 组件之外，还有额外的 ClickCounterButton 组件(无状态和无逻辑)。

ClickCounterButton 没有自己的 `onClick` 事件处理程序(也就是没有 `this.handler` 或 `this.handleClick`)，而使用父组件通过属性(`this.props.handler`)传递给它的处理程序。通常，使用这种方法对于处理按钮中的事件是有益的，因为按钮是无状态的展示/木偶组件。可以在其他用户界面中重复使用此按钮。

代码清单 6.7 展示了渲染按钮的展示组件的代码(ch06/onclick-props/jsx/click-counter-button.jsx)。渲染这个元素的父组件 Content 非常简短，如代码清单 6.8 所示。

#### 代码清单 6.7 无状态按钮组件

```
class ClickCounterButton extends React.Component {  
  render() {  
    return <button  
      onClick={this.props.handler}  
      className="btn btn-danger">  
      Increase Volume (Current volume is {this.props.counter})  
    </button>  
  }  
}
```

图 6.11 所示的 ClickCounterButton 是一个比傻瓜<sup>1</sup>还傻瓜的组件，但是这对于架构而言是有益的，因为组件简单且容易掌握。

<sup>1</sup> [www.imdb.com/title/tt0109686](http://www.imdb.com/title/tt0109686)

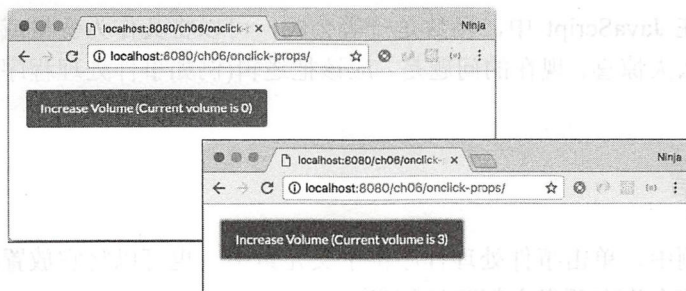


图 6.11 将事件处理程序作为属性传递给按钮(展示组件)可以使按钮标签中的计数器增加，它也是按钮的属性

ClickCounterButton 组件还使用了 counter 属性，该属性通过 `{this.props.counter}` 渲染。如果还记得第 2 章的例子，那么应该知道为子组件提供像 ClickCounterButton 这样的属性非常简单。使用标准属性语法 `name=VALUE` 即可。例如，要为 ClickCounterButton 组件提供 counter 和 handler 属性，只需要在父组件(这里指 Content 组件)的 `render()` 方法的 JSX 定义中指定属性即可：

```
<div>
  <ClickCounterButton
    counter={this.state.counter}
    handler={this.handleClick}/>
</div>
```

在 ClickCounterButton 中，counter 是一个属性，因此也是不可变的。但在父组件 Content 中，它是一个 state 并且是可变的(关于属性和状态的更新，请参阅第 4 章)。很明显，名称可以不同，当把属性传递给子组件时，不必保留这些名称。但我发现，保持相同的名称可以帮助理解不同组件之间的数据相关性。

发生什么事情了？初始 counter(状态)在父组件 Content 中被设置为 0。事件处理程序也定义在这里。因此，子组件(ClickCounterButton)触发了父组件上的事件。父组件 Content 拥有 `constructor()` 和 `handleClick()`，如代码清单 6.8 所示(ch06/onclickprops/jsx/content.jsx)。

代码清单 6.8 以属性方式传递事件处理程序

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)
    this.state = {counter: 0}
  }
  handleClick(event) {
    this.setState({counter: ++this.state.counter})
  }
  render() {
    return (
      <div>
        <ClickCounterButton
          counter={this.state.counter}
          handler={this.handleClick}/>
      </div>
    )
  }
}
```

在 `constructor()` 中绑定上下文，因此可以直接使用 `this.setState()`，this 指代 Content 类的实例



如前所述, 在 JavaScript 中, 函数是一等公民, 可以将其作为变量或属性进行传递。因此这应该没什么大惊喜。现在的问题是, 应该把逻辑(例如事件处理程序放在子组件还是父组件中呢?

### 6.1.6 组件通信

在前面的示例中, 单击事件处理程序位于父元素中。也可以将它放置在子元素中, 但是放在父元素中将允许在两者之间交换信息。

以按钮为例, 但这次将从 `render()`(1、2、3 等)中移除计数器的值。组件是单一、纯展现的(还记得吗?), 所以计数器会在另一个组件 `Counter` 中。因此, 现在一共有三个组件: `ClickCounterButton`、`Content` 和 `Counter`。

如图 6.12 所示, 现在有两个组件: 按钮及其内部的文本。它们都有来自父组件 `Content` 的属性(state)。与前面的例子相比(见图 6.11), 这里需要在按钮和文本之间进行通信, 以便记录单击次数。换言之, `ClickCounterButton` 和 `Counter` 需要相互通信。它们将通过 `Content` 来实现, 而不是直接通信(直接通信是一种不好的模式, 因为会导致紧耦合)。

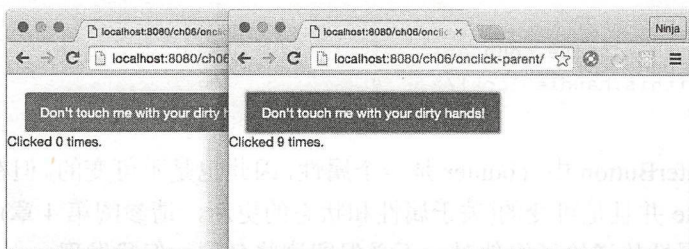


图 6.12 分割状态, 并使用两个无状态子组件(通过允许它们借助父组件进行数据交换): 一个用于计数(文本), 另一个用于按钮

`ClickCounterButton` 和前面的例子一样仍然是无状态组件, 就像大多数 React 组件应有的样子: 只有属性和 JSX, 参见代码清单 6.9。

代码清单 6.9 按钮组件使用来自 `Content` 的事件处理程序

```
class ClickCounterButton extends React.Component {  
  render() {  
    return <button  
      onClick={this.props.handler}  
      className="btn btn-info">  
      Don't touch me with your dirty hands!  
    </button>  
  }  
}
```

当然, 也可以将 `ClickCounterButton` 作为函数而不是类来编写, 如下所示:

```
const ClickCounterButton = (props) => {  
  return <button  
    onClick={props.handler}  
    className="btn btn-info">
```



```

    Don't touch me with your dirty hands!
  </button>
}

```

下面的新组件 **Counter** 展示了代表计数器的 **value** 属性(名称可以不同, 并不总是需要使用 **Counter**):

```

class Counter extends React.Component {
  render() {
    return <span>Clicked {this.props.value} times.</span>
  }
}

```

最后, 我们来看看提供属性的父组件: 一个是事件处理程序, 另一个是计数器。需要更新渲染参数, 但是其余代码仍然保持完整, 参见代码清单 6.10(ch06/onclick-parent/jsx/content.jsx)。

代码清单6.10 给两个组件传递事件处理程序和状态

```

class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleClick = this.handleClick.bind(this)
    this.state = {counter: 0}
  }
  handleClick(event) {
    this.setState({counter: ++this.state.counter})
  }
  render() {
    return (
      <div>
        <ClickCounterButton handler={this.handleClick}/>
        <br/>
        <Counter value={this.state.counter}/>
      </div>
    )
  }
}

```

要回答在何处放置事件处理逻辑的初始问题, 经验法则是: 如果需要子组件之间的交互, 就将其放到父组件或包装组件中。如果事件只关心子组件, 则不需要使用事件处理程序来对父组件链上的其他组件进行污染。

## 6.2 响应 React 不支持的 DOM 事件

表 6.1 列出了 React 支持的事件, 可能你还想知道有哪些不支持的 DOM 事件。例如, 假如你的任务是创建一个响应式的 UI, 随着窗口大小(resize 事件)的调整, 它需要变得更大或更小。但 React 不支持这个事件, 有一种方法可以捕获 **resize** 和任何其他事件。其实你已经知道用来实现它的 React 特性: 生命周期事件。

在本例中，将实现单选按钮。如你所知，标准的 HTML 单选按钮元素在不同浏览器中是不一致的(更大或更小)。出于这个原因，我在 DocuSign 工作时，开发了一种可伸缩的 CSS 单选按钮(<http://mng.bz/kPMu>)来替代标准的 HTML 单选按钮。我使用 jQuery 来实现，这些 CSS 按钮可以通过 jQuery 来操作它们的 CSS 以实现缩放。来看看如何在 React 中创建可伸缩的单选按钮 UI。当调整屏幕大小时，将会使用 React 做出相同的 CSS 按钮缩放，如图 6.13 所示。

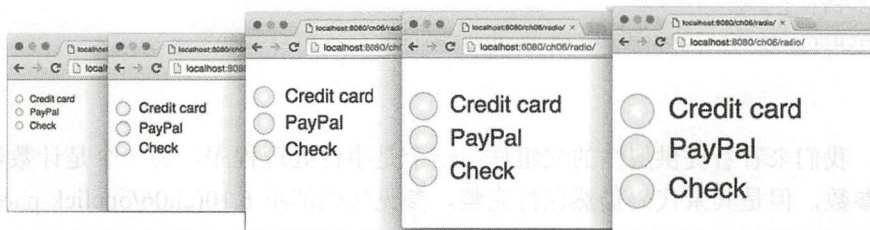


图 6.13 React 实现的可伸缩 CSS 单选按钮，可以监听窗口的 resize 事件。

当窗口的尺寸发生变化时，单选按钮的大小也随之改变

如前所述，React 不支持 resize 事件，按如下所示把它添加到元素中并不会起作用：

```
...
render() {
  return <div>
    <div onResize={this.handleResize}
      className="radio-tagger"
      style={this.state.taggerStyle}>
    ...
```

有一种简单的方法可以绑定一些不受支持的事件(比如 `resize`)和需要支持的大多数自定义元素：使用 React 组件生命周期事件。代码清单 6.11(ch06/radio/jsx/radio.jsx)在 `componentDidMount()` 中为 `window` 对象添加了 `resize` 事件监听器，并在 `componentWillUnmount()` 中将其移除，以确保组件从 DOM 中卸载时不会留下任何东西。在组件被移除之后仍然保留事件监听器，是引入内存泄漏的一种非常好的方式，它可能会在某个时候使应用崩溃。相信我，内存泄漏很可能会为你带来一个喝着饮料、失眠、眼睛红肿的夜晚，用于调试代码。

#### 代码清单 6.11 使用生命周期事件来监听 DOM 事件

```
class Radio extends React.Component {
  constructor(props) {
    super(props)
    this.handleResize = this.handleResize.bind(this)
    let order = props.order
    let i = 1
    this.state = {
      outerStyle: this.getStyle(4, i),
      innerStyle: this.getStyle(1, i),
      selectedStyle: this.getStyle(2, i),
      taggerStyle: {top: order*20, width: 25, height: 25}
    }
  }
```

在状态中  
保存样式

```

    }
    getStyle(i, m) {
      let value = i*m
      return {
        top: value,
        bottom: value,
        left: value,
        right: value,
      }
    }
    componentDidMount() {
      window.addEventListener('resize', this.handleResize)
    }
    componentWillUnmount() {
      window.removeEventListener('resize', this.handleResize)
    }
    handleResize(event) {
      let w = 1+ Math.round(window.innerWidth / 300)
      this.setState({
        taggerStyle: {top: this.props.order*w*10, width: w*10, height: w*10},
        textStyle: {left: w*13, fontSize: 7*w}
      })
    }
    ...
  }

```

使用一个函数来创建多个样式，这个函数使用宽度和乘数来计算

绑定React不支持的一个事件侦听器到window对象上

从window对象上移除React不支持的事件监听器

基于新的屏幕尺寸，实现一个神奇的函数来处理单选按钮的尺寸调整

辅助函数 `getStyle()` 抽象了一些样式，因为 CSS 中有重复，例如 `top`、`bottom`、`left` 和 `right`，但它们的值取决于窗口的宽度。因此，`getStyle()` 接收宽度值和乘数 `m`，并返回像素值(React CSS 中的数字会变成像素)。

其余的代码很简单。需要做的就是实现 `render()` 方法，该方法使用状态和属性来渲染四个 `<div/>` 元素。每个 `<div/>` 元素都有一种在 `constructor()` 中定义过的特殊样式，如代码清单 6.12 所示。

代码清单6.12 为样式使用状态中的值以便缩放元素

```

...
render() {
  return <div>
    <div className="radio-tagger" style={this.state.taggerStyle}>
      <input type="radio" name={this.props.name} id={this.props.id}>
    </input>
    <label htmlFor={this.props.id}>
      <div className="radio-text" style={this.state.textStyle}>{this.props.
        label}</div>
      <div className="radio-outer" style={this.state.outerStyle}>
        <div className="radio-inner" style={this.state.innerStyle}>
          <div className="radio-selected" style={this.state.selectedStyle}>
            </div>
          </div>
        </div>
      </div>
    </label>
  </div>
</div>
}

```



这就是 Radio 组件的实现。本例的要点是，通过在组件中使用生命周期事件，可以创建自定义事件监听器。在本例中，使用了 window 对象。这类似于 React 的事件监听器的工作方式：将事件绑定到 document 对象上，就像在本章开头那样，并且不要忘记在 unmount 事件中移除事件监听器。

如果感兴趣的是可伸缩的单选按钮和它们的非 React 实现(jQuery)，我在 <http://mng.bz/kPMu> 上写了一篇单独的博文，并且创建了一个在线示例，参见 <http://jsfiddle.net/DSYz7/8>。当然，你也可以在本书的源代码中找到 React 版本的实现。

这使我们想到了与其他 UI 库(如 jQuery)进行集成的话题。

## 6.3 React 和其他库的集成：jQuery UI 事件

如你所见，React 提供了标准的 DOM 事件，但是当集成另一个库(这个库使用(触发或监听)非标准事件)时，需要些什么呢？例如，假设有一个使用 slide 元素(比如在滑竿控制元素内)的 jQuery 组件，你希望将一个 React 小部件集成到 jQuery 应用中。可以通过使用生命周期事件 componentDidMount 和 componentWillUnmount 来绑定任何 React 没有提供的 DOM 事件。

正如你对生命周期事件持有的猜测，当组件被挂载时绑定事件监听器，当组件被卸载时移除事件监听器。移除事件监听器(可以看作一次清理)很重要，所以没有任何事件监听器会被遗弃并在冲突和性能上造成问题(孤立的事件处理程序是没有 DOM 节点的处理程序，它们会产生潜在的内存泄漏)。

例如，假设你在一家音乐流媒体公司工作，你的任务是在新版本的网络播放器上实现音量控制(想想 Spotify 和 iTunes)。除了已存在的 jQuery slider 元素(<http://plugins.jquery.com/ui.slider>)，还需要添加一个标签和两个按钮。

你希望实现一个带有数字的标签，以及两个按钮来加 1 和减 1。它们在一起的工作方式是：当用户左右滑动滑块(滑竿上的方块)时，数值和按钮上显示的值应该相应地变化。同样，用户应该可以单击任意按钮，滑块也应该相应地左右移动。本质上，不仅要创建一个滑块，还要创建图 6.14 所示的小部件。



图 6.14 React 组件(按钮和文本“Value:...”)可以和其他库集成，例如 jQuery Slider，从而让不同库的不同元素之间相互通信

### 6.3.1 集成按钮

在集成方便，至少有两个选择：首先，可以在 React 组件中为 jQuery Slider 绑定事件；其次，可以使用 window 对象。我们从使用第一种方法集成按钮开始。

注意：这种集成按钮的方法是紧耦合的。对象互相依赖。一般来说，应该避免紧耦合。

另一种更松散耦合的方法将在我们使用第一种方法后实现, 用来集成标签。

当 jQuery 滑块上出现 slide 事件(意味着值发生了变化)时, 需要更新按钮上的值(按钮上的文本)。可以将事件监听器绑定到 componentDidMount 中的 jQuery slider, 并在触发 slide 事件时触发 React 组件(handleSlide)上的一个方法。随着每一次滑动和值的改变, 将会更新 state 值(sliderValue)。SliderButtons 实现了这个方法, 如代码清单 6.13 所示(ch06/slider/jsx/slider-buttons.jsx)。

代码清单6.13 通过事件与jQuery插件集成

```
class SliderButtons extends React.Component {
  constructor(props) {
    super(props)
    this.state = {sliderValue: 0}
  }
  handleSlide(event, ui) {
    this.setState({sliderValue: ui.value})
  }
  handleChange(value) {
    return () => {
      $('#slider').slider('value', this.state.sliderValue + value)
      this.setState({sliderValue: this.state.sliderValue + value})
    }
  }
  componentDidMount() {
    $('#slider').on('slide', this.handleSlide)
  }
  componentWillUnmount() {
    $('#slider').off('slide', this.handleSlide)
  }
  ...
}
```

设置初始值为0

定义一个方法以便在按钮被单击时更新滑块

使用jQuery方法设置新值

jQuery会传递两个参数: 一个jQuery事件对象和一个包含当前值的ui对象, ui对象被用来更新state值

为-1和+1按钮使用工厂函数模式

把state更新成新的值

卸载时移除事件监听器

SliderButtons 的 render() 方法有两个带 onClick 事件的按钮; 一个动态的 disabled 属性, 以便其值既不会小于 0(如图 6.15 所示), 也不会超出 100; 以及用于按钮的 Twitter Bootstrap 类, 参见代码清单 6.14(ch06/slider/jsx/slider-button.jsx)。

代码清单6.14 渲染slider按钮

```
...
render() {
  return <div>
    <button disabled={this.state.sliderValue < 1 ? true : false}
      className="btn default-btn"
      onClick={this.handleChange(-1)}>
      1 Less ({this.state.sliderValue-1})
    </button>
    <button disabled={this.state.sliderValue > 99 ? true : false}
      className="btn default-btn"
      onClick={this.handleChange(1)}>
      1 More ({this.state.sliderValue+1})
    </button>
  </div>
}
```

使用-1调用this.handleChange 以便从工厂函数获取对应的函数

使用className属性应用Twitter Bootstrap类

使用三目运算符在数值大于99或小于1时禁用按钮

将slider的新值渲染为按钮的标签



最终结果是，如果值小于或大于设定的范围(0~100)，按钮将被禁用。例如，当值为 0 时，Less 按钮被禁用，如图 6.15 所示。

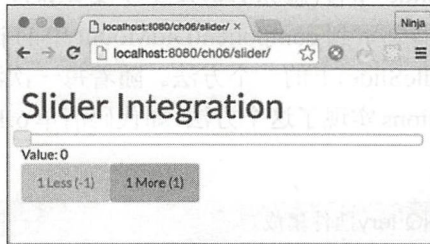


图 6.15 以编程公式禁用 Less 按钮以阻止负值产生

拖动滑块可以更改按钮上的文本并且根据需要禁用/启用按钮。要感谢 `handleChange()` 方法中对 `slider()` 方法的调用，单击按钮可以使滑块左右移动。接下来，将要实现标签组件，它是一个 `SliderValue` React 组件。

### 6.3.2 集成标签

我们已经了解了在 React 方法中如何直接调用 jQuery。与此同时，还可以通过另一个对象来捕获事件，实现 jQuery 和 React 的解耦。这是一种松散耦合模式，通常是可取的，因为它有助于避免额外的依赖。换言之，不同组件无须知道彼此的实现细节。因此，`SliderValueReact` 组件将不知道如何调用 jQuery `Slider`。这样非常好，因为以后可以更容易地从 `Slider` 迁移到拥有不同接口的 `Slider 2.0`。

可以通过 jQuery 将事件绑定到 `window` 对象上，并且在 React 组件生命周期方法中定义 `window` 对象的事件监听器。下面的代码清单 6.15 展示了 `SliderValue` 组件(ch06/slider/jsx/slider-value.jsx)。

代码清单 6.15 通过 window 对象集成 jQuery 插件

```
class SliderValue extends React.Component {
  constructor(props) {
    super(props)
    this.handleSlide = this.handleSlide.bind(this)
    this.state = {sliderValue: 0}
  }
  handleSlide(event) {
    this.setState({sliderValue: event.detail.ui.value})
  }
  componentDidMount() {
    window.addEventListener('slide', this.handleSlide)
  }
  componentWillUnmount() {
    window.removeEventListener('slide', this.handleSlide)
  }
  render() {
    return <div className="" >
      Value: {this.state.sliderValue}
    </div>
  }
}
```

绑定 slide 事件监听器到 window 对象上以便可以触发 handleSlide()

从 window 对象上移除 slide 事件监听器，以防止产生孤立的事件监听器和内存泄漏



另外, 还需要分发一个自定义事件。在第一种方法(SliderButtons)中, 不需要这样做, 因为使用了现有的插件事件。在这个实现中, 必须创建一个携带数据的事件并将其分发到 window 对象上。可以在创建 jQuery Slider 对象的代码旁实现 slide 自定义事件调度器, 该调度器是 index.html(ch06/slider/index.html)中的 script 标签。

代码清单6.16 在jQuery UI插件上设置事件监听器

```

为jQuery Slider创建一个事件处理程
序, 它会派发一个自定义事件
    let handleChange = (e, ui)=>{
        var slideEvent = new CustomEvent('slide', {
            detail: {ui: ui, jQueryEvent: e}  )
        })
        window.dispatchEvent(slideEvent)
    }
    $('#slider').slider({
        'change': handleChange,
        'slide': handleChange
    })

```

传递包含了最新滑块值的jQuery数据对象

创建一个自定义事件

派发一个事件给window对象

使用一个id为slider的容器创建一个滑块

为change(编程)和slide(UI)事件绑定事件监听器

运行代码, 按钮和标签将无缝工作。这里使用了两种方法: 一种是松散耦合, 另一种是紧密耦合。后者的实现更简短, 但前者是可取的, 因为将来修改代码将会更容易。

从这个集成案例中可以看到, React 可以通过在 `componentDidMount()` 中监听事件来和其他类库良好地配合工作。React 以一种非排它的方式运作。React 能与其他库轻松集成是一个巨大的优势, 因为开发人员可以逐渐切换到 React 而不是从头重写整个应用, 甚至可以和 React 一起无限期地使用钟爱的老式类库。

## 6.4 测验

1. 选择正确的事件声明语法: `onClick=this.doStuff`、`onclick={this.doStuff}`、`onClick="this.doStuff"`、`onClick={this.doStuff}` 或 `onClick={this.doStuff() }`。
2. 在 React 组件中声明的 `componentDidMount()` 方法在服务器端渲染期间不会被触发, 这么理解正确还是错误?
3. 一种在父组件和子组件之间传递信息的方式是将对象从子组件移到父组件上, 这么理解正确还是错误?
4. 默认情况下, 可以在事件处理程序之外异步地使用 `event.target`, 这么理解正确还是错误?
5. 可以通过在 React 组件生命周期事件中设置事件监听器来集成第三方库和 React 不支持的事件, 这么理解正确还是错误?

## 6.5 小结

- `onClick` 用于捕获鼠标和触控板的单击操作。

- 事件监听器的 JSX 语法是 `<a onNAME={this.METHOD}>`。
- 如果要在事件处理程序中使用 `this` 指代组件类实例的值，可以在 `constructor()` 或 JSX 中使用 `bind()` 绑定事件处理程序
- `componentDidMount()` 只在浏览器端触发。`componentWillMount()` 会在浏览器和服务端两端触发。
- React 通过提供合成事件对象来支持大多数标准的 HTML DOM 事件。
- `componentDidMount()` 和 `componentWillUnmount()` 可以用来集成其他框架和 React 不支持的事件。

## 6.6 测验答案

1. `onClick={this.doStuff}` 是正确的，因为传给 `onClick` 的只能是函数定义而不是调用(确切地说，是调用的结果)。

2. 正确。`componentDidMount()` 仅在浏览器端 React 而不是服务器端 React 中执行。这就是开发人员使用 `componentDidMount()` 进行 AJAX/XHR 请求的原因。有关组件生命周期事件，请参阅第 5 章。

3. 正确。将数据上移到组件树层次结构中，可将其传递给不同的子组件。

4. 错误。这个对象会被重用，所以不能在异步操作中使用，除非在 `SyntheticEvent` 上调用 `persist()`。

5. 正确。组件生命周期事件是执行此操作的最佳位置之一，因为它们允许在组件处于活动状态之前和被移除之前执行准备工作。

# 第 7 章

## 在 React 中使用表单

本章内容：

- 定义表单和表单元素
- 捕获数据变化
- 使用引用来访问数据
- 在表单元素中捕获用户输入数据的替代方法
- 为表单元素设置默认值

到目前为止，我们已经学习了事件、状态、组件构成和 React 中的其他重要话题、特征和概念，但是抛开用户事件，我们尚未学习如何捕获输入的文本以及通过其他表单元素(例如 `input`、`textarea` 和 `option`)获得的输入。在 Web 开发中使用它们非常重要，因为它们使得应用可以接收用户数据(例如文本)和行为(例如单击)。

本章涉及迄今为止涵盖的所有内容，你将会看到所有这些是如何配合在一起的。

### 7.1 在 React 中使用表单的最佳实践

一般在 HTML 中，当我们使用 `input` 元素时，页面的 DOM 节点包含 `input` 元素的值。可以通过 `document.getElementById('email').value` 方法或 jQuery 方法来获取元素的值。实际上，DOM 就是存储空间。

在 React 中，当使用表单或其他用户可输入的字段时，例如单个的文本域或按钮，有一个需要注意和解决的问题。React 文档提到：“React 组件必须任何时候都保持视图与



状态的一致，而不仅仅是在初始化时。” React 保持如下简单的逻辑：根据声明的样式来绘制 UI。React 是这么描述 UI 的：它的最终阶段决定了它该表现出什么样子。

发现差异了吗？对于传统的 HTML 表单元素，表单元素的状态变化与用户输入同步。但是，React 使用声明式方法来描述 UI。输入需要动态反映到 state 属性中。

因此，选择不维护组件的状态(在 JavaScript 中)并且不与视图进行同步会带来问题；可能会有一种情况，内部的状态与视图不一致。React 不了解状态的变化。这会导致各种各样的麻烦和错误，从而否定 React 的简单哲学。最佳实践是保持 React 的 render() 方法尽量和真实的 DOM——那些包含数据的表单元素——紧密相关。

考虑下面的文本输入域。React 在 render() 方法中必须包含该组件的最新值。因此，需要使用 value 更新该元素的值。但是，如果在 HTML 中使用了 <input> 元素，React 将会保持 render() 与真实 DOM 同步。React 不允许用户改变此值。尝试一下。这确实有些奇怪，但是对 React 却是合适的。

```
render() {  
  return <input type="text" name="title" value="Mr." />  
}
```

这段代码对于任何状态都会描绘出同样的视图。所以，input 的值会一直是“Mr。”。另一方面，当用户单击或输入时，input 必须响应更改。基于这几点，让值动态变化。这是更好的实现，因为组件会根据状态进行更新：

```
render() {  
  return <input type="text" name="title" value={this.state.title} />  
}
```

但 state 的值是什么？React 无法通过表单元素获取用户的输入。需要实现 onChange 事件处理程序以捕获变更：

```
handleChange(event) {  
  this.setState({title: event.target.value})  
}  
render() {  
  return <input type="text" name="title" value={this.state.title}  
    onChange={this.handleChange.bind(this)} />  
}
```

考虑到这些，最佳实践是实现内部状态与视图的同步(见图 7.1)：

- ① 在 render() 中使用来自 state 的值定义元素。
- ② 使用 onChange 捕获表单元素的变更。
- ③ 在事件处理程序中更新内部状态。
- ④ 将新值保存在 state 中，然后重新执行 render() 以更新视图。

乍看起来好像有很多工作要做，但是当越来越多地使用 React 时，便会越欣赏这种方式。它被称为单向绑定，因为状态改变了视图，就是这样。没有反向：只有从状态到视图的单向数据流动。使用单向绑定，一个库不会自动更新(组件的)状态(或模型)。单向绑定的主要优点之一是会消除复杂性，尤其是开发大型应用程序时，多个视图需要更新多个状态

(数据模型), 反之亦然(见图 7.2)。

简单并不意味着编写更少的代码。例如这个例子甚至需要编写更多的代码来手动设置数据, 使数据流从事件处理程序到状态(会被渲染到视图); 但是当涉及复杂的 UI 和具有大量状态和视图的单页面应用时, 这种方法往往更加优越。简单并不代表容易。

与单向绑定相反, 双向绑定使视图可以自动更改状态, 而无须显式实现该过程。双向绑定正是 Angular 1 的工作机制。有趣的是, Angular 2 借鉴了 React 的单向绑定理念, 并将其设为默认(仍然可以明确地使用双向绑定)。

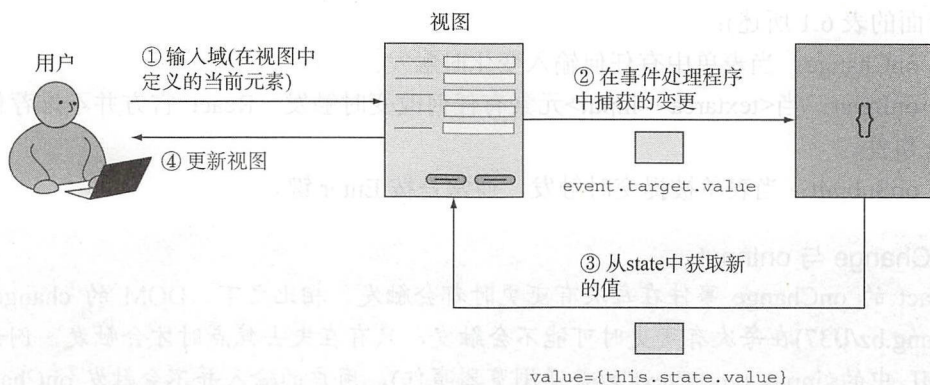


图 7.1 使用表单元素的正确方式: 由用户输入到事件, 再到状态和视图

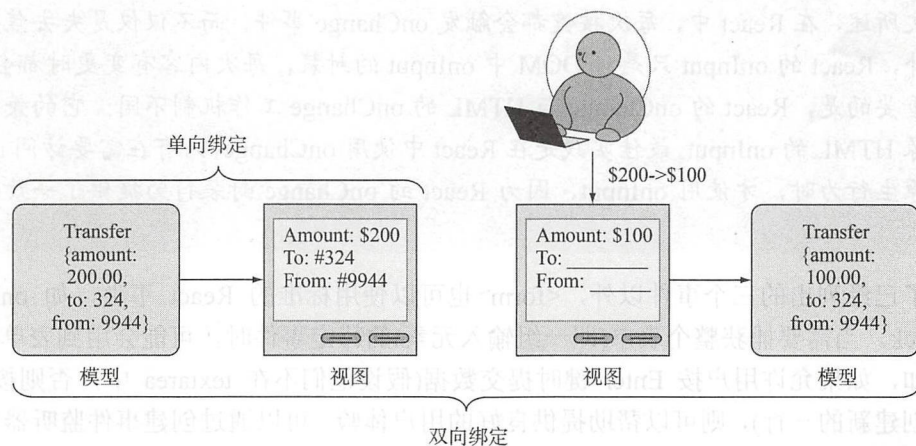


图 7.2 单向绑定负责模型到视图的转换, 双向绑定也将视图的更改反映到模型

因此, 将首先介绍使用表单的最佳实践。这被称为受控组件, 并且它能确保组件内部状态始终与视图同步。受控表单元素之所以被称为受控组件, 是因为 React 负责控制或设置它的值。相应的另一种是非受控组件, 将在 7.2 节中讨论。

我们已经学习了在 React 中使用输入字段的最佳实践: 捕获变更并将其同步到状态, 如图 7.1 所示(输入改变视图)。接下来, 我们来看看如何定义表单及其元素。

### 7.1.1 在 React 中定义表单及响应事件

我们从<form>元素开始。通常, 我们不希望输入元素随意放在 DOM 中。如果有很多



功能不同的输入集合，情况会变得更加糟糕。替代方法是：将作用相同的输入元素包裹在 `<form></form>` 元素中。

使用 `<form>` 封装并不是必需的。在简单的 UI 中单独使用表单元素也很好。在更复杂的 UI 中，可能在单个页面上有多组元素，对于每一组使用这样的表单是明智的。React 的 `<form>` 与 HTML 的 `<form>` 在渲染上类似，所以针对 HTML 表单的任何规范也适用于 React 的 `<form>` 元素。例如，根据 HTML5 规范，表单不能嵌套<sup>1</sup>。

`<form>` 元素可以注册事件。除了标准的 React DOM 事件外，React 还支持三个表单事件(如前面的表 6.1 所述)：

- `onChange`：当表单中有任何输入变化时触发。
- `onInput`：当 `<textarea><input>` 元素有任何改变时触发。React 官方并不推荐使用此事件。
- `onSubmit`：当表单被提交时触发，通常是按 Enter 键。

### onChange 与 onInput

React 的 `onChange` 事件在每次有变更时都会触发，相比之下，DOM 的 `change` 事件(<http://mng.bz/1J37>)在每次有变更时可能不会触发，只有在失去焦点时才会触发。例如，对于 HTML 中的 `<input type="text">`(普通浏览器事件)，用户的输入并不会触发 `onChange` 事件，只有当用户按下 Tab 键或用鼠标单击其他元素(失去焦点)时，才会触发 `onChange` 事件。正如前文所述，在 React 中，每次按键都会触发 `onChange` 事件，而不仅仅是失去焦点才触发。另外，React 的 `onInput` 只是对 DOM 中 `onInput` 的封装，每次内容有变更时都会触发。

最重要的是，React 的 `onChange` 与 HTML 的 `onChange` 工作机制不同：它的兼容性更好，更像 HTML 的 `onInput`。最佳实践是在 React 中使用 `onChange`，只有在需要访问 `onInput` 事件的原生行为时，才使用 `onInput`，因为 React 的 `onChange` 封装行为提供了一致性，更加理性。

除了已经列出的三个事件以外，`<form>` 也可以使用标准的 React 事件，如 `onKeyUp` 和 `onClick`。当需要捕获整个表单(即一组输入元素)的特定事件时，可能会用到表单事件。

例如，如果允许用户按 Enter 键时提交数据(假设他们不在 `textarea` 中，否则按 Enter 键应该创建新的一行)，则可以帮助提供良好的用户体验。可以通过创建事件监听器来监听表单提交事件，并在事件监听器中触发 `this.handleSubmit()`：

```
handleSubmit(event) {  
  ...  
}  
  
render() {  
  <form onSubmit={this.handleSubmit}>  
    <input type="text" name="email" />  
  </form>  
}
```

注意：需要在 `render()` 之外实现 `handleSubmit()` 函数，就像处理其他事件一样。React

<sup>1</sup> HTML5 规范要求，内容必须是流内容，但 `<form>` 元素不是，详见 [www.w3.org/TR/html5/forms.html#the-form-element](http://www.w3.org/TR/html5/forms.html#the-form-element)



不需要命名规则，所以可以使用任何希望的名称来命名事件处理程序，尽量保持容易理解和一致性。本书坚持使用最流行的原则：给事件处理程序添加前缀 `handle`，以便与常规的类方法区分开来。

注意：提醒一点，在设置事件处理程序时，不要调用方法(不要放圆括号)，并且不要在花括号旁使用双引号(正确写法：`EVENT={this.METHOD}`)。对于一些人来说，这是基本的 JavaScript 语法，并且很明显。但是，我在 React 代码中看到的有关这两个错误的次数让人难以置信：传递的是函数的定义而不是结果；并且使用花括号作为 JSX 属性的值。

实现按 Enter 键提交表单的另一种方法是手动监听 `keyup` 事件(`onKeyUp`)，并检查键值码(Enter 键的键值码是 13)：

```
handleKeyUp(event) {
  if (event.keyCode == 13) return this.sendData()
}
render() {
  return <form onKeyUp={this.handleKeyUp}>
    ...
  </form>
}
```

需要注意的是，需要在类/组件的其他位置实现 `sendData()` 方法。此外，还需要在 `constructor()` 中使用 `bind(this)` 将 `sendData()` 绑定到类的上下文。

总而言之，可以在表单元素上创建事件，而不是在表单中的元素上单独创建。接下来，我们将介绍如何定义表单元素。

### 7.1.2 定义表单元素

只需要四个元素就可以实现 HTML 中所有的输入字段：`<input>`、`<textarea>`、`<select>` 和 `<option>`。还记得在 React 中，属性是不可变的吗？但是，表单元素是特殊的，因为用户需要与元素交互并且改变这些属性。对于所有其他元素，则没有此特性。

React 通过给它们设置可变属性——`value`、`checked` 和 `selected`——来实现此特性。这些特殊的可变属性也称为交互式属性。

注意：React DOM 也支持其他与构建表单相关的元素，例如 `<keygen>`、`<datalist>`、`<fieldset>` 和 `<label>`。这些元素不具有像可变属性 `value` 之类的超能力，它们被渲染为相应的 HTML 标签。因此，本书只关注这四个有超能力的重点元素。

在添加到表单元素的 `onChange` 等事件中，可以读到以下具有交互能力的属性/字段(见 6.1.3 节)：

- `value`：适用于 `<input>`、`<textarea>` 和 `<select>`。
- `checked`：适用于 `<input>` 中的 `type="checkbox"` 和 `type="radio"`。
- `selected`：适用于 `<option>` (与 `<select>` 一起使用)。

可以通过这些交互式(可变)属性读取这些值并改变它们。我们来看一些关于如何定义每种元素的例子。

## <input>元素

<input>元素根据 type 属性的值渲染出不同的字段：

- text: 纯文本输入字段。
- password: 带有屏蔽显示功能的文本输入字段(用于隐私)。
- radio: 单选按钮。使用相同的名称创建一组单选按钮。
- checkbox: 复选框。使用相同的名称创建一组复选框。
- button: 按钮表单元素。

所有<input>类型元素——checkbox 和 radio 除外——的主要用法是使用 value 作为元素的可变属性。例如，email 输入字段可以使用 email 元素的 state 和 onChange 事件处理程序：

```
<input
  type="text"
  name="email"
  value={this.state.email}
  onChange={this.handleEmailChange}/>
```

没有用 value 作为主要可变属性的两个例外是 checkbox 和 radio，它们使用 checked，因为对于这两种类型，每个 HTML 元素都有一个值，并且该值不会改变，但是 checked/selected 的状态会改变。例如，可以定义三个 radio 为一组(radioGroup)，如图 7.3 所示。

如前所述，value 是固定的，因为不需要改变这些值。随用户行为变化的是元素的 checked 属性，如代码清单 7.1 所示(ch07/elements/jsx/content.jsx)。

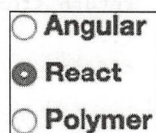


图 7.3 单选按钮组

### 代码清单7.1 渲染单选按钮并处理更改

```
class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleRadio = this.handleRadio.bind(this)
    ...
    this.state = {
      ...
      radioGroup: {
        angular: false,
        react: true,
        polymer: false
      }
    }
  }
  handleRadio(event) {
    let obj = {} // erase other radios
    obj[event.target.value] = event.target.checked // true
    this.setState({radioGroup: obj})
  }
  ...
  render() {
    return <form>
      <input type="radio"
        name="radioGroup"
        value='angular'
        checked={this.state.radioGroup['angular']}>
```

在state中设置单选按钮  
为默认选中

使用target.checked属性获取一个布  
尔值，该布尔值指示此单选按钮是  
否被选中

使用state对象的属性  
或任何state属性

```

        onChange={this.handleRadio}/>
        <input type="radio"
          name="radioGroup"
          value='react'
          checked={this.state.radioGroup['react']}
          onChange={this.handleRadio}/>
        <input type="radio"
          name="radioGroup"
          value='polymer'
          checked={this.state.radioGroup['polymer']}
          onChange={this.handleRadio}/>
        ...
      </form>
    }
  }
}

```

使用相同的onChange事件处理程序，因为可以从target.value获取单选按钮值

对于复选框，可以遵循类似于单选按钮的方法：使用 checked 属性和 state 中的布尔值，这些布尔值可以存储在 checkboxGroup 的 state 中：

```

class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleClickbox = this.handleClickbox.bind(this)
    // ...
    this.state = {
      // ...
      checkboxGroup: {
        node: false,
        react: true,
        express: false,
        mongodb: false
      }
    }
  }
}

```

然后事件处理程序(在 constructor() 中绑定)抓取当前值，根据 event.target.checked 的值为对象的 event.target.value 添加 true 或 false，并设置状态：

```

handleClickbox(event) {
  let obj = Object.assign(this.state.checkboxGroup)
  obj[event.target.value] = event.target.checked
  this.setState({checkboxGroup: obj})
}

```

← true或false

由于 radio 只能有一个选定的值，因此不需要从状态中赋值。因此，可以使用一个空对象。复选框不是这样，它们可以选择多个值，所以需要合并对象而不是替换对象。

在 JavaScript 中，对象使用引用传递和赋值。所以在语句 obj = this.state.checkboxGroup 中，obj 是真实的状态。再次强调，不能直接改变状态。为了避免任何潜在的冲突发生，最佳实践是使用 Object.assign() 合并赋值。这种技术也叫作 clone。另外，表现欠佳但是更具技巧性的方法是使用 JSON 赋值：

```

clonedData = JSON.parse(JSON.stringify(originalData))

```



当在 `state` 中使用数组替代对象, 并且需要合并赋值时, 使用 `clonedArray=Array.from (originArray)` 或 `clonedArray = originArray.slice()`。

可以使用 `handleCheckbox()` 事件处理程序来从 `event.target.value` 取值。下面的代码清单 7.2 展示了使用 `state` 的值设置四个复选框的 `render()` 方法(`ch07/elements/jsx/content.jsx`), 如图 7.4 所示。

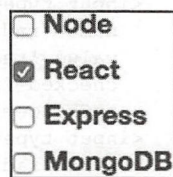


图 7.4 使用 React 作为预选项渲染复选框

#### 代码清单 7.2 定义复选框

```
<input type="checkbox"
  name="checkboxGroup"
  value='node'
  checked={this.state.checkboxGroup['node']}
  onChange={this.handleCheckbox}/>
<input type="checkbox"
  name="checkboxGroup"
  value='react'
  checked={this.state.checkboxGroup['react']}
  onChange={this.handleCheckbox}/>
<input type="checkbox"
  name="checkboxGroup"
  value='express'
  checked={this.state.checkboxGroup.express}
  onChange={this.handleCheckbox}/>
<input type="checkbox"
  name="checkboxGroup"
  value='mongodb'
  checked={this.state.checkboxGroup['mongodb']}
  onChange={this.handleCheckbox}/>
```

将`state`作为值。它可以是对象的属性或只是`state`属性

使用`onChange`捕获操作

当键值是有效的JS名称时, 使用点符号

由于在构造函数中已经绑定了, 因此不需要再次在元素中绑定

实际上, 当使用复选框或单选按钮时, 可以对每个元素中的值进行硬编码, 并使用 `checked` 作为可变属性。让我们来看看如何使用其他输入元素。

#### <textarea>元素

`<textarea>` 元素用于捕获和显示长文本输入, 例如笔记、博客帖子、代码片段等。在常规 HTML 中, `<textarea>` 使用内部的 HTML (即子节点) 作为值:

```
<textarea>
  With the right pattern, applications...
</textarea>
```

图 7.5 展示了一个例子。

正确的模式可以使应用程序更具扩展性、更容易维护。

如果你渴望有一天成为一名 Node.js 架构师 (也许你已经成功了, 并且想扩展一些知识), 那么这个例子正适合你。

图 7.5 定义并渲染 `<textarea>`

相反, React 使用 `value` 属性。鉴于此, 在 `<textarea>` 内部设置一个 HTML 节点或文本节点作为值会违反这种模式。React 会用默认值(更多关于默认值的讨论将在 7.2.4 节中介绍)覆盖 `<textarea>` 的子节点(如果使用了子节点):

```
<!-- Anti-pattern: AVOID doing this! -->
<textarea name="description">{this.state.description}</textarea>
```

但是, 建议使用 `<textarea>` 的 `value` 属性:

```
render() {
  return <textarea name="description" value={this.state.description}/>
}
```

要监听变更, 像 `<input>` 元素一样, 使用 `onChange`。

### `<select>`和`<option>`元素

`select` 和 `option` 字段具有良好的用户体验, 允许用户从预先填充的列表中选择单个值或多个值。元素列表紧紧隐藏在元素后面, 直到用户展开(在单次选择的情况下), 如图 7.6 所示。

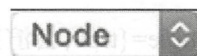


图 7.6 渲染并设置下拉列表的值

`<select>`是 React 中另一个与常规 HTML 行为相异的元素。例如, 通常在 HTML 中, 会使用 `selectDOMNode.selectedIndex` 或 `selectDOMNode.selectedOptions` 来获取被选中元素的引用。在 React 中, 需要为 `<select>` 提供 `value`, 如代码清单 7.3 所示(ch07/elements/jsx/content.jsx)。

#### 代码清单7.3 渲染表单元素

```
...
constructor(props) {
  super(props)
  this.state = {selectedValue: 'node'}
}
handleSelectChange(event) {
  this.setState({selectedValue: event.target.value})
}
...
render() {
  return <form>
    <select
      value={this.state.selectedValue}
      onChange={this.handleSelectChange}>
      <option value="ruby">Ruby</option>
      <option value="node">Node</option>
      <option value="python">Python</option>
    </select>
  </form>
}
```

这段代码渲染了一个下拉菜单, 并设置了 `node` 值(必须在 `constructor()` 中设置, 如图 7.6

所示)。Node 很赞！

有时需要用到复选元素,在 JSX/React 中,可以通过给 `multiple` 属性不设置任何值(React 默认是 `true`)或设置值 `{true}` 来实现。

提示: 请注意,为保持一致性和避免混淆,建议将所有的布尔值都包装在花括号 `{}` 而不是双引号 `"` 中。当然, `"true"` 和 `{true}` 被视为相同的结果。但是 `"false"` 也会被判定为 `true`, 这是因为在 JavaScript 中,字符串 `"false"` 被视为 `true`。

要预选多项,可以将一组选项传递给 `<select>` 的 `value` 属性。例如,下面的代码预选了 Meteor 和 React:

```
<select multiple={true} value={['meteor', 'react']}>
  <option value="meteor">Meteor</option>
  <option value="react">React</option>
  <option value="jQuery">jQuery</option>
</select>
```

`multiple={true}` 会渲染复选元素,并且 Meteor 和 React 的值已经被预选,如图 7.7 所示。

总的来说,在 React 中定义表单元素与定义常规 HTML 中的表单元素没有太大不同,只是需要更频繁地使用 `value` 属性。这种相似性让人愉悦。但是,定义只是工作的一半,另一半是捕获这些值。在前面的例子中已经做了这一点。我们来详细了解事件的捕获。

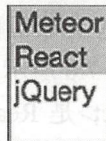


图 7.7 预选并渲染复选元素

### 7.1.3 捕获表单变更

如前所述,需要设置 `onChange` 事件监听器来捕获表单元素的变更。此事件取代了常规 DOM 的 `onInput`。换句话说,如果需要 `onInput` 的常规 HTML DOM 行为,可以使用 React 的 `onInput`。另一方面,React 的 `onChange` 与常规 DOM 的 `onChange` 不完全相同。只有当元素失去焦点时,常规的 DOM `onChange` 才可能触发,而 React 的 `onChange` 会在所有新的输入上触发。具体哪些触发 `onChange`,每种元素各有所异:

- `<input>`、`<textarea>` 和 `<select>`: 当 `value` 改变时触发 `onChange`。
- `<input>` 的 `checkbox` 和 `radio` 类型: `checked` 改变时触发 `onChange`。

基于此映射,读取值的方法各不相同。`SyntheticEvent` 对象将作为事件处理程序的参数,此对象有 `value`、`checked` 或 `selected` 等属性,因元素而异。

为监听变更,可以在组件中的某些位置定义事件处理程序(也可以在 JSX 的 `{}` 中内联定义),并创建组件的 `onChange` 属性以指向事件处理程序。例如,代码清单 7.4 从 `email` 字段捕获变更(`ch07/elements/jsx/content.jsx`)。

#### 代码清单 7.4 渲染表单元素并捕获变更

```
handleChange(event) {
  console.log(event.target.value)
}
```



```
render() {
  return <input
    type="text"
    onChange={this.handleChange}
    defaultValue="hi@azat.co"/>
}
```

有趣的是，如果没有定义 `onChange` 但提供了值，React 会抛出警告并将元素设为只读。如果目标是只读字段，最好使用 `readOnly` 明确加以定义。这不仅会消除警告，也会确保阅读此代码的其他开发人员知道这是一个设计为只读的字段。要显式地设置值，请将 `readOnly` 的值设置为 `{true}`——`readOnly={true}`——或者添加 `readOnly` 属性，而不设置值，React 会默认将该属性的值设置为 `true`。

捕获到元素中的变更后，可以将这些变更存储在组件的状态中：

```
handleChange(event) {
  this.setState({emailValue: event.target.value})
}
```

这些信息迟早都会被发送到服务器或另一个组件，这种情况下，可以将状态中的值整齐有序地组织起来。

例如，假设要创建包含用户姓名、地址、电话号码和社会安全号码的贷款申请表。每个输入字段都处理自己的变更。在表单底部，将设置提交按钮，将状态值发送到服务器。代码清单 7.5 给出了上述字段，`input` 字段的 `onChange` 函数会维护自身的状态(`ch07/elements/jsx/content.jsx`)。

代码清单7.5 渲染表单元素

```
constructor(props) {
  super(props)
  this.handleInput = this.handleInput.bind(this)
  this.handleSubmit = this.handleSubmit.bind(this)
  ...
}
handleFirstNameChange(event) {
  this.setState({firstName: event.target.value})
}
...
handleSubmit() {
  fetch(this.props['data-url'], {method: 'POST', body:
    JSON.stringify(this.state)})
    .then((response) => {return response.json()})
    .then((data) => {console.log('Submitted: ', data)})
}
render() {
  return <form>
    <input name="firstName"
      onChange={this.handleFirstNameChange}
      type="text"/>
    ...
  </form>
}
```

捕获`firstName`字段的变更，并保存在状态中

使用浏览器的基于`promise`的`Fetch`方法(作为实验性的写法，但是大部分现代浏览器均支持)将数据发送到`data-url`属性中的URL

```
    <input
      type="button"
      onClick={this.handleSubmit}
      value="Submit"/>
  </form>
}
```

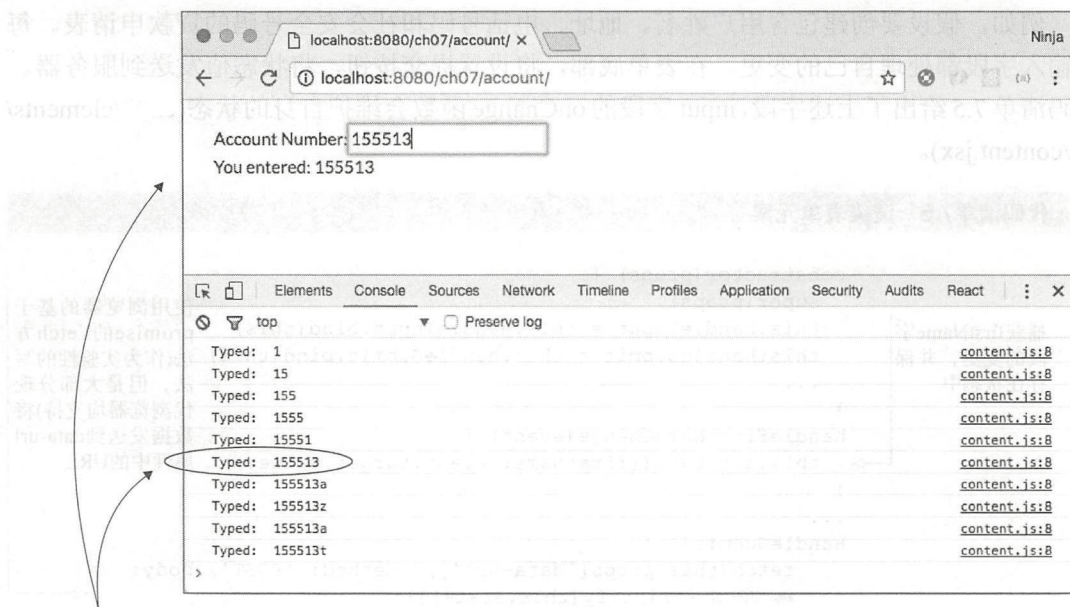
定义事件处理程序来响应提交按钮

注意: Fetch 是一种实验性的本地浏览器方法, 可以基于 promise 执行 AJAX / XHR 请求。可以在 <http://mng.bz/mbMe> 上阅读相关用法和支持情况(大多数现代浏览器都支持)。我们已经学习了如何定义元素、使用事件捕获变更并且更新状态(用于显示值)。

### 7.1.4 账户字段示例

继续分析贷款申请的情况, 一旦贷款获得批准, 用户需要输入他们希望贷款转账的账户。现在可以使用新技能实现一个账户输入组件。这是一个可控元素, 是在 React 中处理表单的最佳实践。

在代码清单 7.6(ch07/account/jsx/content.jsx)所示的组件中, 账号输入字段只接受数字(见图 7.8)。要限制输入为数字(0~9), 可以使用受控组件来清除所有非数字值。事件处理程序只有在对输入过滤后才会设置状态。



只允许数字, 因为React  
控制着元素的值

图 7.8 可以接受任何输入值, 但是正如控制台所示, 只允许使用数字作为值并回显, 因为这是一个受控组件

代码清单7.6 实现一个受控组件

```

class Content extends React.Component {
  constructor(props) {
    super(props)
    this.handleChange = this.handleChange.bind(this)
    this.state = {accountNumber: ''}
  }

  handleChange(event) {
    console.log('Typed: ', event.target.value)
    this.setState({accountNumber: event.target.value.replace(/[^\d-]/ig,
    })

  }

  render() {
    return <div>
      Account Number:
      <input
        type="text"
        onChange={this.handleChange}
        placeholder="123456"
        value={this.state.accountNumber}/>
      <br/>
      <span>{this.state.accountNumber.length > 0 ? 'You entered: ' +
        this.state.accountNumber: ''}</span>
    </div>
  }
}

```

设置账户的初始值为空字符串

输出未经过滤的输入值

过滤值并设置状态

捕获变更

通过设置状态控制元素

如果账户非空，就输出。length是字符串属性，可以返回字符的数量。如果value为空，不输出

使用正则表达式(<http://mng.bz/r7sq>)/[^\d-]/ig 和字符串的 `replace`(<http://mng.bz/2Qon>)方法来移除所有非数字字符。正则表达式 `replace(/[^\d-]/ig, "")` 不是很复杂，它用空字符串替换除数字以外的所有字符。ig 表示不区分大小写和全局查找(换句话说，匹配所有字符)。

`render()` 含有输入字段，它是一个受控组件，因为 `value={this.state.accountNumber}`。当尝试此例时，只能输入数字，因为 React 将新的状态设置为仅包含数字的过滤后的值(见图 7.9)。

通过遵循在 React 中使用输入元素和表单的最佳实践，可以实现验证，并确保它们表现出应用程序所希望的样子。

注意：显然，在账户组件中实现前端验证，不会阻止黑客将恶意数据通过 XHR 请求发送到服务器。因此，确保对后端/服务器和/或业务层进行了合适的验证，如 ORM 或 ODM ([https://en.wikipedia.org/wiki/Object-relational\\_mapping](https://en.wikipedia.org/wiki/Object-relational_mapping))。

至此，你已经了解了使用表单的最佳实践：创建受控组件。让我们来看一些替代方案。





React 开发者工具显示了元素的结构、属性和状态，在本例中，状态仅限于数字

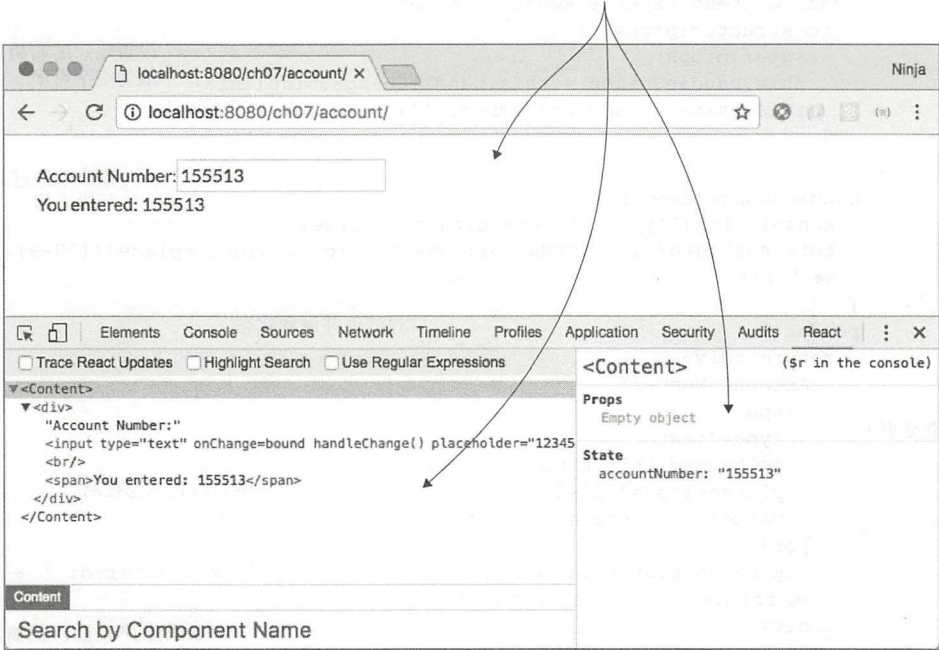


图 7.9 受控元素通过将状态设置为仅支持数字来过滤输入

## 7.2 使用表单的其他方式

使用受控表单是最佳实践。但是如前所述，这种方法需要做一些额外的工作，因为需要手动捕获变更和更新状态。实际上，如果使用字符串、属性或状态定义了属性 value、checked、selected 的值，那么组件将受控(由 React 控制)。

同时，若未设置属性 value 的值(既非状态也非静态值)，表单元素可能不受控制。由于本章开头列出的原因(视图的 DOM 状态可能与 React 的内部状态不同)，并不推荐这样做。当构建将提交到服务器的简单表单时，非受控元素可能很有用。换言之，当建立一个没有大量突变和用户行为的简单 UI 元素时，考虑使用非受控模式；这是一种骇客技术，在大多数时候都应该避免。

通常，要使用非受控组件，需要定义表单提交事件，典型的例子有按钮上的 onClick 或表单上的 onSubmit。一旦定义了事件处理程序，就可以有如下两个选择：

- 像受控元素一样捕获变更，并使用状态提交而不是用元素值提交(毕竟，这是一种非受控方法)。
- 不捕获变更。

第一种方法很简单。这一类基本上拥有相同的事件监听器并且会更新 state。如果仅在最后阶段使用状态(用于表单提交)，那么编码就太多了。



**警告：**React 还比较新，最佳实践仍在对实际应用程序的开发和维护中不断形成。维护规范可能会在维护大型 React 应用程序的几年后发生变化。非受控组件是尚未有明确共识的灰色区域。可能有说法，称这是一种反模式，应该完全避免。我不会选择站队，但是会提供足够的信息使你能够做出自己的判断。我这样做是因为，我相信你有足够可用的知识和明智的行动选择。基准是：考虑阅读本章的其余可选部分——可能使用也可能不使用的工具。

### 7.2.1 可捕获变更的非受控元素

如你所见，在 React 中，非受控组件意味着 value 属性不是由 React 库设置的。发生这种情况时，组件的内部值(或状态)可能与组件表示(或视图)中的值不同。基本上，内部状态和表现之间是不一致的。组件状态可以有一些逻辑(例如验证)；并且使用非受控组件模式，视图将接受表单元素中的任何用户输入，从而造成视图和状态之间的差异。

例如，如下文本输入框为非受控元素，因为 React 并未设置值：

```
render() {  
  return <input type="text" />  
}
```

用户的任何输入会立即渲染到视图中。这么做好还是坏？少安毋躁，我将详述此种情形。

要捕获非受控组件的变更，可以使用 onChange。例如，图 7.10 中的输入字段有 onChange 事件处理程序(this.handleChange)、一个引用(textbook)和一个占位符，当字段为空时，它将产生一个灰色的文本框。

代码清单 7.7 中的 handleChange() 方法将值打印在控制台中并使用 event.target.value 更新状态(ch07/uncontrolled/jsx/content.jsx)。

代码清单 7.7 非受控组件捕获变更

```
class Content extends React.Component {  
  constructor(props) {  
    super(props)  
    this.state = {textbook: ''}  
  }  
  
  handleChange(event) {  
    console.log(event.target.value)  
    this.setState({textbook: event.target.value})  
  }  
  
  render() {  
    return <div>  
      <input  
        type="text"  
        onChange={this.handleChange}  
        placeholder="Eloquent TypeScript: Myth or Reality" />  
      <br/>  
      <span>{this.state.textbook}</span>  
    </div>  
  }  
}
```

设置内部值为空字符串

当输入字段有变化时更新状态

只有事件监听器，并不设置input元素的值

使用<span>输出state变量，state变量是在handleChange()方法中设置的





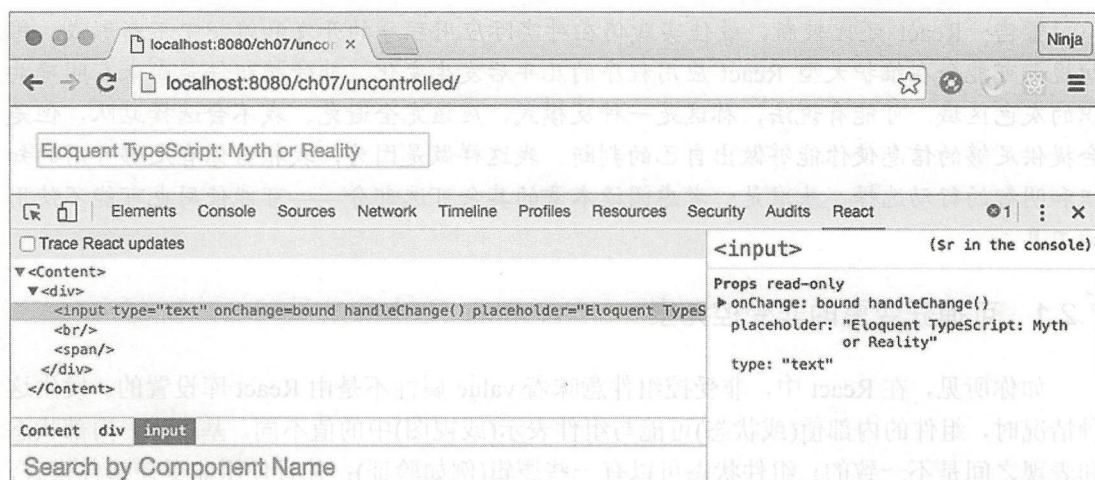


图 7.10 此非受控组件在应用程序中没有被赋值

这个思路是：用户可以输入他们想要的任何内容，因为 React 不会控制输入字段的值。React 要做的所有工作只是捕获新的值(onChange)并设置状态。反过来，更改状态会更新<span>(见图 7.11)。

用这种方法，可以为输入字段实现一个事件处理程序。能完全跳过事件捕获吗？

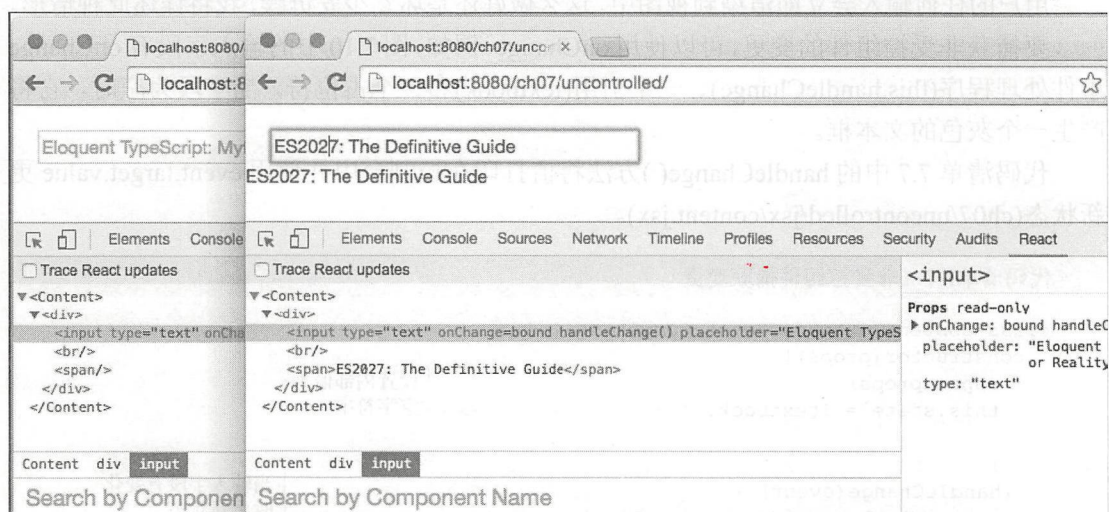


图 7.11 由于捕获了变更，输入更新了状态，但是 DOM 文本输入元素的值非受控

## 7.2.2 不捕获变更的非受控元素

我们来看看第二种方法。当想要使用它们(例如，表单提交)时，问题是：所有值是否准备就绪。在捕获变更的方法中，可以拥有状态中的所有数据。当选择不捕获非受控元素的变更时，数据仍在 DOM 中。要想在 JavaScript 对象中获取数据，解决方案是使用引用，如图 7.12 所示。请对比图 7.12 中非受控元素如何工作与图 7.1 中展示受控组件功能的受控元素流。





注意：当使用受控组件或非受控组件捕获数据时，数据一直保存在状态中。本节讨论的方法并非如此。

总而言之，为了使用不捕获变更的非受控元素这样的方式，需要一种方法来访问其他元素以从中获取数据。

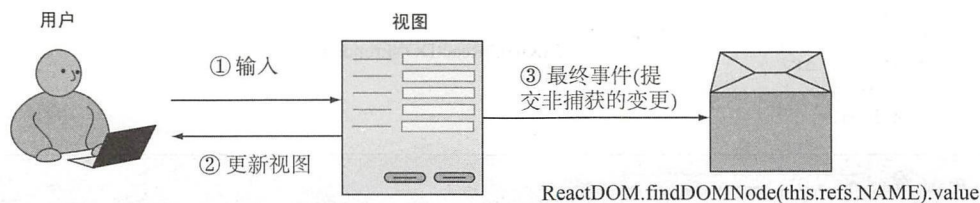


图 7.12 使用非受控元素但不捕获变更，而是通过引用访问值

### 7.2.3 使用引用获取值

在使用非受控的、不捕获事件(如 `onChange`)的组件时，可以使用引用访问值，但引用不是此模式独有的。虽然使用引用作为反模式不被推荐，仍可以在任何合适的场景中使用引用。原因是当 `React` 元素被正确定义时，每个元素使用内部状态与视图的状态(DOM)同步，对引用的需求几乎不存在。但是需要理解引用，所以在这里介绍它们。

通过引用，可以获取 `React` 组件的 DOM 元素(或节点)。当需要获取表单元素的值时，这很方便，但是不会捕获元素中的变更。

要使用引用，需要做以下两件事：

- 确保 `render()` 方法返回的元素有以驼峰风格命名的 `ref` 属性(例如 `email:<input ref="userEmail" />`)
- 在其他方法中访问 DOM 实例的已命名引用。例如，在事件处理程序中，`this.refs.NAME` 变为 `this.refs.userEmail`。

`this.refs.NAME` 可以提供 `React` 组件的实例，但是如何获取值？DOM 节点更有用。可以通过调用 `ReactDOM.findDOMNode(this.refs.NAME)` 访问组件的 DOM 节点：

```
let emailNode = ReactDOM.findDOMNode(this.refs.email)
let email = emailNode.value
```

我发现这个方法写起来有些笨拙(太冗长)，所以记得可以使用别名：

```
let fD = ReactDOM.findDOMNode
let email = fD(this.refs.email).value
```

考虑图 7.13 中的示例，该例捕获用户电子邮件地址和评论。在浏览器控制台中输出值。这个项目结构和其他项目结构截然不同。看起来像下面这样：



```

/email
/css
  bootstrap.css
/js
  content.js
  react.js
  react-dom.js
  script.js
/jsx
  content.jsx
  script.jsx
index.html

```

主要组件与编译的脚本

JSX中的ReactDOM.render() 语句

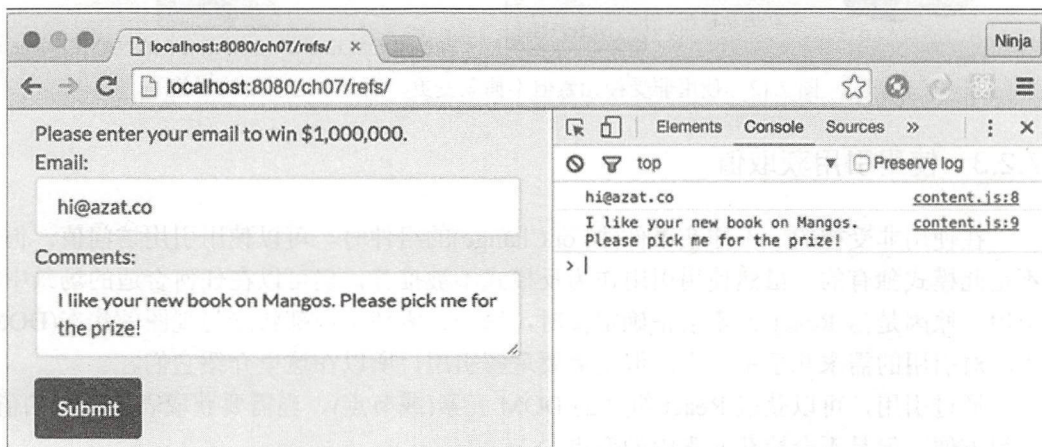


图 7.13 非受控表单，从两个字段获取数据并将它们打印到控制台中

单击提交按钮时，可以访问 `emailAddress` 和 `comments` 的引用并将值输出到两条日志中，如代码清单 7.8 所示(ch07/email/jsx/content.jsx)。

#### 代码清单 7.8 email表单的开始

```

class Content extends React.Component {
  constructor(props) {
    super(props)
    this.submit = this.submit.bind(this)
    this.prompt = 'Please enter your email to win $1,000,000.'
  }
  submit(event) {
    let emailAddress = this.refs.emailAddress
    let comments = this.refs.comments
    console.log(ReactDOM.findDOMNode(emailAddress).value)
    console.log(ReactDOM.findDOMNode(comments).value)
  }
}

```

定义一个类属性

使用引用访问并打印email地址

接下来是托管的 `render()` 函数，它使用 Twitter Bootstrap 类来设计引入的表单，参见代码清单 7.9(ch07/ email/jsx/content.jsx)。记得使用 `className` 设置 `class` 属性！



代码清单7.9 email表单的render()方法

```

render: function() {
  return (
    <div className="well">
      <p>{this.prompt}</p>
      <div className="form-group">
        Email: <input ref="emailAddress" className="form-control"
          type="text" placeholder="hi@azat.co"/>
      </div>
      <div className="form-group">
        Comments: <textarea ref="comments" className="form-control"
          placeholder="I like your website!"/>
      </div>
      <div className="form-group">
        <a className="btn btn-primary" value="Submit"
          onClick={this.submit}>Submit</a>
      </div>
    </div>
  )
}

```

实现用于电子邮件的输入字段，其中包含placeholder元素属性。placeholder属性提供一种视觉辅助，以显示要输入的内容。使用元素的className和ref属性

打印Content组件的prompt属性的值

使用onClick事件编码提交按钮，该事件会调用this.submit

常规的 HTML DOM 节点<textarea>使用 innerHTML 作为值。如前所述，在 React 中可以使用此元素的 value 属性：

```
ReactDOM.findDOMNode(comments).value
```

这是因为 React 实现了 value 属性。这只是 React 为表单元素提供更一致的 API 后所获得的良好特性之一。同时，由于 ReactDOM.findDOMNode() 方法返回一个 DOM 节点，因此也可以访问其他常规 HTML 属性(如 innerHTML)和方法(如 getAttribute())。

现在，可以从组件的几乎任何方法访问元素及其值，而不仅仅是该特定元素的事件处理程序。另外，引用仅适用于使用非受控组件的罕见情形。过度使用引用不是最佳实践。大多数情况下，在受控元素中都不需要使用引用，因为可以使用组件的状态。

也可以为 JSX 中的 ref 属性分配一个函数。这个函数在组件挂载时只调用一次。在该函数中，可以将 DOM 节点保存在一个实例的属性中，如 this.emailInput：

```

<input ref={(input) => { this.emailInput = input }}
  className="form-control"
  type="text"
  placeholder="hi@azat.co"/>

```

非受控组件需要的编码较少(更新状态和捕获变更都是可选的)，但是它们会引发另一个问题：不能把值设置为状态或硬编码值，因为这样做会约束元素(例如，不能使用 value={this.state.email})。如何设置初始值？假设贷款申请已部分填写并保存，用户恢复会话，需要显示已经填写的信息，但不能使用 value 属性。我们来看看如何设置默认值。

## 7.2.4 默认值

假设希望贷款应用程序使用现有数据预先填充某些字段。在常规的 HTML 中，可以定



义一个带 `value` 属性的表单字段，用户可以修改页面上的元素。但是 React 使用 `value`、`checked` 和 `selected` 来保持元素内部状态和视图间的一致性。在 React 中，如果像下面这样强制赋值：

```
<input type="text" name="new-book-title" value="Node: The Best Parts"/>
```

得到的将是一个只读输入字段。大多数情况下，这不符合需求。因此，在 React 中，使用特殊属性 `defaultValue` 设置该值并允许用户修改表单元素。

例如，假设表单已经保存，并且要为用户回填 `<input>` 字段。这种情况下，需要为表单元素使用 `defaultValue` 属性。可以像下面这样设置输入字段的初始值：

```
<input type="text" name="new-booktitle"
      defaultValue="Node: The Best Parts"/>
```

如果使用 `value` 属性(`value="JSX"`)代替 `defaultValue`，这个元素会变为只读。不仅变为受控元素，而且当用户在 `<input>` 元素中输入时，值也不会改变，如图 7.14 所示。这是因为值是固定的，而 React 会维护该值。这可能不是你所希望的。显然，在现实应用中，以编程方式获取值，在 React 中意味着使用属性(`this.props.name`)：

```
<input type="text" name="new-book-title" defaultValue={this.props.title}/>
```

或使用状态：

```
<input type="text" name="new-book-title" defaultValue={this.state.title}/>
```



图 7.14 将值设置为字符串时，`<input>` 元素的值在网页上显示为冻结(不可更改)

React 的 `defaultValue` 属性常用于非受控组件；但是，和引用一样，默认值可以与受控组件或其他任何方案一起使用。在受控组件中不需要使用默认值，因为可以在 `constructor()` 的 `state` 中定义这些值，例如 `this.state = { defaultName: 'Abe Lincoln'}`。

由此可见，大多数关于 UI 的工作都是用便捷的表单元素完成的。要使它们既美观，又易于理解和使用，还必须维护错误消息的良好交互体验、前端验证以及其他不起眼的功能，例如工具提示、可扩展的单选按钮、默认值和提示语句。构建 UI 可能很复杂并且迅速失控！幸运的是，React 通过让表单元素使用跨浏览器 API，使工作变得更轻松。

## 7.3 测验

1. 非受控组件设置了值，而受控组件没有设置值，对吗？
2. 设置默认值的正确语法是下面哪个？`default-value`、`defaultValue` 还是 `defVal`？
3. React 团队推荐使用 `onChange` 而不是 `onInput`，对吗？
4. 使用下面哪一项设置文本域的值？子组件、`innerHTML` 还是 `value`？
5. 在表单中，`selected` 适用于下面哪一项？`<input>`、`<textarea>` 还是 `<option>`？
6. 下面哪种是获取 DOM 节点引用的最佳方式？`React.findDOMNode(this.refs.email)`、

`this.refs.email`、`this.refs.email.getDOMNode`、`ReactDOM.findDOMNode(this.refs.email)`还是 `this.refs.email.getDomNode`?

## 7.4 小结

- 表单的首选方式是使用受控组件，其中事件监听器捕获并存储状态中的数据。
- 使用非受控组件，无论是否捕获变更都是骇客行为，应该避免。
- 引用和默认值可以用在任何组件上，但是受控组件一般不需要。
- React 的 `<textarea>` 使用 `value` 属性而非内部内容。
- `this.refs.NAME` 是一种访问类的引用的方法。
- `defaultValue` 允许为元素设置视图(DOM)的初始值。
- 定义引用的方式为 `ref="NAME"`。

## 7.5 测验答案

1. 错误。受控组件/元素的定义是：它们设置了值。
2. `defaultValue`。其他选项都是无效的名称。
3. 正确。在常规 HTML 中，`onChange` 并不会因为每次变更都触发；但在 React 中，每次变更都会触发。
4. 在 React 中，通过给 `value` 设置值来保持一致性；但在 HTML 中，是使用 `innerHTML`。
5. `<option>`。
6. 使用 `ReactDOM.findDOMNode(reference)`或回调(在列表中没有正确答案)。

# 第 8 章

## 扩展 React 组件

### 本章内容：

- 给组件设置默认属性
- 了解 React 属性类型和验证
- 渲染子节点
- 创建可复用的高阶组件
- 最佳实践：展示和容器组件

到目前为止，我们已经介绍了如何创建组件并使它们具有交互性，以及允许用户输入(事件和输入元素)。使用这些知识来构建使用 React 组件的网站，还有很长的路要走。当依赖其他软件工程师(开源贡献者或同事)创建的组件时，某些烦恼会不断出现，对大型项目尤为如此。

例如，当使用他人编写的组件时，如何知道是否提供了正确的属性？另外，如何为现有组件(也适用于其他组件)添加一些功能？这些都是开发的可扩展性问题：如何扩展代码？即代码库越来越大时，如何使用代码？React 中的某些功能和模式可以帮助解决这些问题。

如果了解如何有效地构建复杂的 React 应用程序，这些话题很重要。例如，高阶组件允许增强组件的功能，属性类型提供了类型安全性检查，且有高可信度。

在本章结尾，你将熟悉 React 的大部分功能。你将擅长使代码变得开发者友好(使用属性类型)，使工作更有效(使用组件名称和高阶组件)。你的队友甚至可能会惊叹于你优雅的解决方案。这些功能将有助于高效使用 React，所以让我们来进一步深入了解吧！



## 8.1 组件中的默认属性

假设正在构建一个 `Datepicker` 组件，该组件需要一些必需的属性，例如行数、区域设置和当前日期：

```
<Datepicker currentDate={Date()} locale="US" rows={4}/>
```

如果新的团队成员尝试使用该组件，但是忘记传递必要的 `currentDate` 属性，会发生什么？或者，当另一名同事传递字符串 `"4"` 而不是数字 `4` 时，又会发生什么呢？该组件将不会执行任何操作(值未定义)，或更糟糕的是：它可能崩溃，你的同事可能会责怪你(是引用错误？)。

遗憾的是，这是 Web 开发中的常见情况，因为 JavaScript 是一种松散类型的语言。幸运的是，React 提供了一个功能，可以为属性设置默认值：`defaultProps` 静态属性。在下一节中我们将继续讨论如何用属性类型标记问题。

`defaultProps` 的主要优点是：如果属性丢失，则会呈现默认值。要在组件类上设置属性的默认值，请定义 `defaultProps`。例如，在上述日期选择组件的定义中，可以添加一个静态类属性(而不是一个实例属性，因为在实例的 `constructor()` 中设置的属性不会生效)：

```
class Datepicker extends React.Component {  
  ...  
}  
Datepicker.defaultProps = {  
  currentDate: Date(),  
  rows: 4,  
  locale: 'US'  
}
```

为了进一步说明 `defaultProps`，假设有一个组件渲染的是按钮。通常，按钮具有标签，但是这些标签需要可定制。如果省略自定义值，最好有默认值。

`buttonLabel` 属性指定按钮的标签，在 `render()` 方法的 `return` 属性中使用。你可能希望此属性始终包含 `Submit`，即使该值不在上述代码中设置。为此，实现 `defaultProps` 静态类属性，该属性使实例对象包含 `buttonLabel` 属性，且 `buttonLabel` 属性带有默认值：

```
class Button extends React.Component {  
  render() {  
    return <button className="btn" >{this.props.buttonLabel}</button>  
  }  
}  
Button.defaultProps = {buttonLabel: 'Submit'}
```

父组件 `Content` 渲染四个按钮，但是其中三个按钮缺少属性：

```
class Content extends React.Component {  
  render() {  
    return (  
      <div>  
        <Button buttonLabel="Start"/>  
        <Button />  
        <Button />  
      </div>  
    )  
  }  
}
```

```
    <Button />
  </div>
)
}
```

你能猜到结果吗？第一个按钮的标签是“Start”，其余按钮的标签是“Submit”（见图 8.1）。

设置默认属性值几乎总是对的，因为这样做使得组件的容错性提高了。换句话说，组件变得更智能，因为即使没有提供任何内容，它们也具有基本的外观和行为。

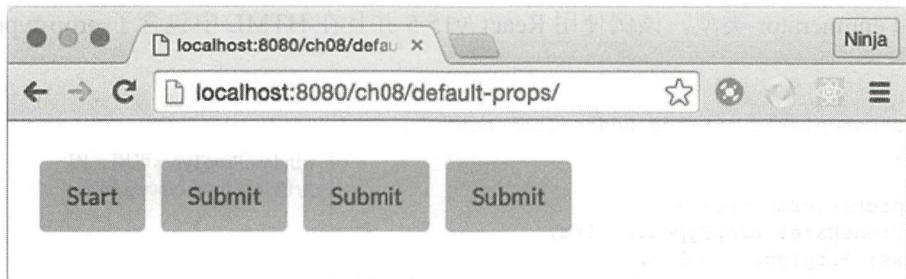


图 8.1 第一个按钮拥有在创建时设置的标签。其他按钮没有，因此返回是默认属性值

下面介绍另一种方式，有默认值意味着可以跳过一遍又一遍地声明初始值。如果大部分时间都使用单个属性值，但仍然希望提供一种修改此值的方法(覆盖默认值)，使用 `defaultProps` 功能就十分合适。覆盖默认值不会引起任何问题，如上述示例中的第一个按钮元素所示。

## 8.2 React 属性类型和验证

回到前面使用 `Datepicker` 组件的例子，不了解属性类型("5"相对于 5)的同事，可以将属性类型设置为与 `React` 组件类一起使用。这可以通过 `propTypes` 静态属性来实现。属性类型的这一功能不会强制改变属性值的数据类型，而是提供警告。也就是说，如果处于开发者模式，并且属性类型不匹配，则会在控制台和产品中收到警告消息；不用做任何处理来防止使用错误的类型。实际上，`React.js` 会在生产模式下抑制此警告。因此，`propTypes` 是一个方便的特性，可以在开发阶段提醒数据类型不匹配。

### React 的生产环境对比开发环境

`React.js` 团队定义开发者模式使用未压缩版本的 `React`，生产模式使用压缩版本。`React` 作者提到：

我们提供两个 `React` 版本：一个用于开发的未压缩版本和另一个用于生产的最小压缩版本。开发版本包括常见错误的额外警告，而生产版本包括额外的性能优化且剥离所有错误消息。

对于 `React v5.5` 及更高版本(本书中大多数示例使用 `React v15.5`)，类型定义来自名为 `prop-types`([www.npmjs.com/package/prop-types](http://www.npmjs.com/package/prop-types))的单独包。需要在 `HTML` 文件中引入

prop-types, 该包将成为一个全局对象(window.PropTypes):

```
<!-- development version -->
<script src="https://unpkg.com/prop-types/prop-types.js"></script>

<!-- production version -->
<script src="https://unpkg.com/prop-types/prop-types.min.js"></script>
```

如果使用 React v5.4 及早期版本, 就没必要单独引入 prop-types, 因为类型都包含在 React 中: React.PropTypes。

下面是一个在 DatePicker 类上定义了静态 propTypes 属性的基本示例, 它使用 string、number 和 enumerator 类型。该例使用 React v15.5 并且在 HTML 中包含了 prop-types(此处未列出):

```
class DatePicker extends React.Component {
  ...
}
DatePicker.propTypes = {
  currentDate: PropTypes.string,
  rows: PropTypes.number,
  locale: PropTypes.oneOf(['US', 'CA', 'MX', 'EU'])
}
```

← window.PropTypes可用, 因为包含了prop-types.js脚本

**警告:** 不要依赖前端的用户输入验证, 因为这可以被轻松绕过。前端验证仅用于更好的用户体验, 应该在服务器端检查所有内容。

要验证属性类型, 请将属性作为键、类型作为值, 放在 propTypes 属性对象中。React.js 的所有类型都在 PropTypes 对象中:

- *PropTypes.string*
- *PropTypes.number*
- *PropTypes.bool*
- *PropTypes.object*
- *PropTypes.array*
- *PropTypes.func*
- *PropTypes.shape*
- *PropTypes.any.isRequired*
- *PropTypes.objectOf(PropTypes.number)*
- *PropTypes.arrayOf(PropTypes.number)*
- *PropTypes.node*
- *PropTypes.instanceOf(Message)*
- *PropTypes.element*
- *PropTypes.oneOfType([PropTypes.number, ...])*

为了演示, 我们通过给默认值添加一些属性类型来增强 defaultProps 示例。该项目的结构类似于 content.jsx、button.jsx 和 script.jsx, 在 index.html 文件中已经包含对 prop-types.js 的引用:



```

<!DOCTYPE html>
<html>

  <head>
    <script src="js/react.js"></script>
    <script src="js/prop-types.js"></script>
    <script src="js/react-dom.js"></script>
    <link href="css/bootstrap.css" type="text/css" rel="stylesheet"/>
    <link href="css/style.css" type="text/css" rel="stylesheet"/>
  </head>

  <body>
    <div id="content" class="container"></div>
    <script src="js/button.js"></script>
    <script src="js/content.js"></script>
    <script src="js/script.js"></script>
  </body>
</html>

```

我们定义一个 `Button` 类，此类带有 `string` 类型的可选标题(title)。要实现它，需要定义一个静态类属性(该类的一个属性)`propTypes`，以 `title` 作为键、`PropTypes.string` 作为值。将下面这段代码放入 `button.js` 中：

```

Button.propTypes = {
  title: PropTypes.string
}

```

也可以设置属性为必需。为此，添加 `isRequired` 作为类型。例如，`title` 属性必需且为 `string` 类型：

```

Button.propTypes = {
  title: PropTypes.string.isRequired
}

```

这个按钮还需要 `handler` 属性，此属性的值必须是一个函数(上一次检查过，没有动作的按钮是没有意义的)。

```

Button.propTypes = {
  handler: PropTypes.func.isRequired
}

```

定义自定义验证也很好。要实现自定义验证，需要做的是创建一条返回 `Error` 实例的语句。然后，在 `propTypes:{...}` 中使用该语句作为属性的值。例如，下面的代码片段使用正则表达式 `emailRegularExpression` 验证了 `email` 属性<sup>1</sup>：

```

...
propTypes = {
  email: function(props, propName, componentName) {
    var emailRegularExpression =

```

---

1 有很多版本的电子邮件验证正则表达式，具体取决于严格性、地区和其他标准。请参阅“Email Address Regular Expression That 99.99% Works”(<http://emailregex.com>)以及“Validate email address in JavaScript?”(Stack Overflow 上的问题，<http://mng.bz/zm37>)和正则表达式库(<http://regexlib.com/Search.aspx?k=email>)

```

/^([\w-]+(?:\.[\w-]+)*)@((?![\w-]+\.)*\w[\w-]{0,6})\.([a-z]{2,6}(?:\.[a-z]{2})?)$/i
  if (!emailRegularExpression.test(props[propName])) {
    return new Error('Email validation failed!')
  }
}
...

```

现在，让我们总结一下。无论是否使用 `title(string)` 和 `handler(必须是函数)` 属性，都会调用 `Button` 组件。代码清单 8.1(ch08/prop-types) 使用属性类型来确保 `handler` 是函数、`title` 是字符串、`email` 遵守提供的正则表达式。

代码清单8.1 使用propTypes和defaultProps

```

class Button extends React.Component {
  render() {
    return <button className="btn">{this.props.buttonLabel}</button>
  }
}

Button.defaultProps = {buttonLabel: 'Submit'}

Button.propTypes = {
  handler: PropTypes.func.isRequired,
  title: PropTypes.string,
  email(props, propName, componentName) {
    let emailRegularExpression =
      /^([\w-]+(?:\.[\w-]+)*)@((?![\w-]+\.)*\w[\w-]{0,6})\.([a-z]{2,6}(?:\.[a-z]{2})?)$/i
    if (!emailRegularExpression.test(props[propName])) {
      return new Error('Email validation failed!')
    }
  }
}

```

handler的值必须是函数

定义可选的title属性，它的值必须是string类型

使用正则表达式定义email验证

接下来，我们实现父组件 `Content`，它渲染 6 个按钮来测试属性类型生成的警告信息，参见代码清单 8.2(ch08/prop-types/jsx/content.jsx)。

代码清单8.2 渲染6个按钮

```

class Content extends React.Component {
  render() {
    let number = 1
    return (
      <div>
        <Button buttonLabel="Start"/>
        <Button />
        <Button title={number}/>
        <Button />
        <Button email="not-a-valid-email"/>
        <Button email="hi@azat.co"/>
      </div>
    )
  }
}

```

触发一个警告，因为title必须是string类型

触发一个警告，因为没有handler

触发一个警告，因为错误的email格式

运行上述代码会导致控制台(不要忘记打开它)显示三条警告消息,如图 8.2 所示。第一个警告是必须指定 handler 函数,我在几个按钮中省略了:

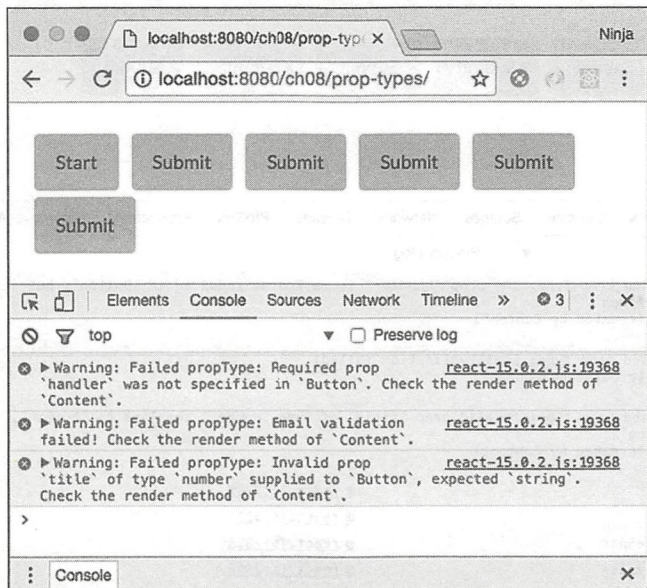


图 8.2 错误的属性类型导致的警告

Warning: Failed propTypes: Required prop `handler` was not specified in `Button`. Check the render method of `Content`.

第二条警告是第四个按钮的 email 格式错误:

Warning: Failed propTypes: Email validation failed! Check the render method of `Content`.

第三条警告是 title 属性的类型错误,类型必须是 string(在一个按钮上提供的是数字):

Warning: Failed propTypes: Invalid prop `title` of type `number` supplied to `Button`, expected `string`. Check the render method of `Content`.

有趣的是,多个按钮缺少 handler,但是只看到一个警告。Content 组件的单个 render() 对每个属性只发出一次警告。

令人欣喜的是,React 会告诉你需要检查哪个父组件。在本例中是 Content 组件。想象一下,如果有几百个组件,这会很有用。

方便的是,如果在 DevTools 中展开信息,可以找到导致警告产生的 Button 元素的行号。在图 8.3 中,展开信息,然后找到文件(content.js)。信息指出这个问题出在第 9 行。

通过在控制台中单击 content.js:9,可以打开该行的 Sources 选项卡,如图 8.4 所示,上面清楚地显示了是什么错误。



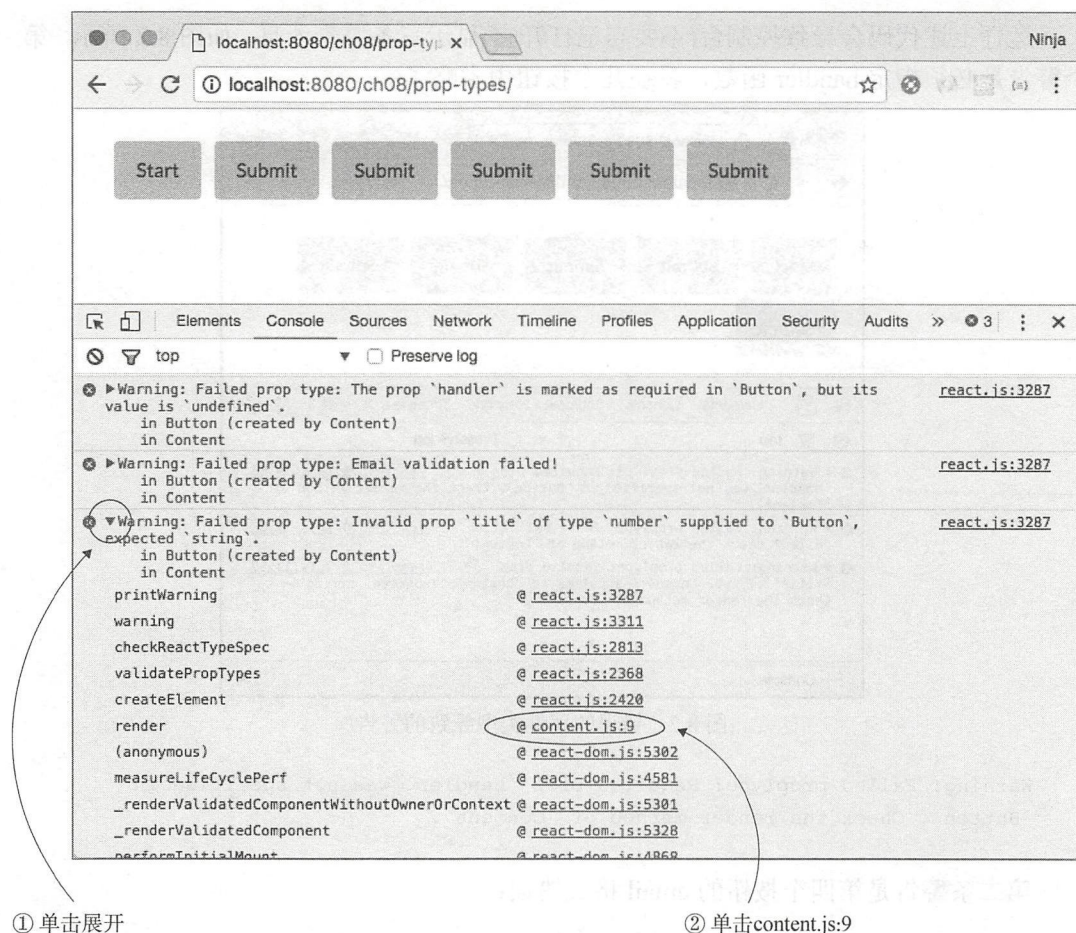


图 8.3 扩展警告以显示有问题的行号: 9

```
React.createElement(Button, { title: number } ),
```

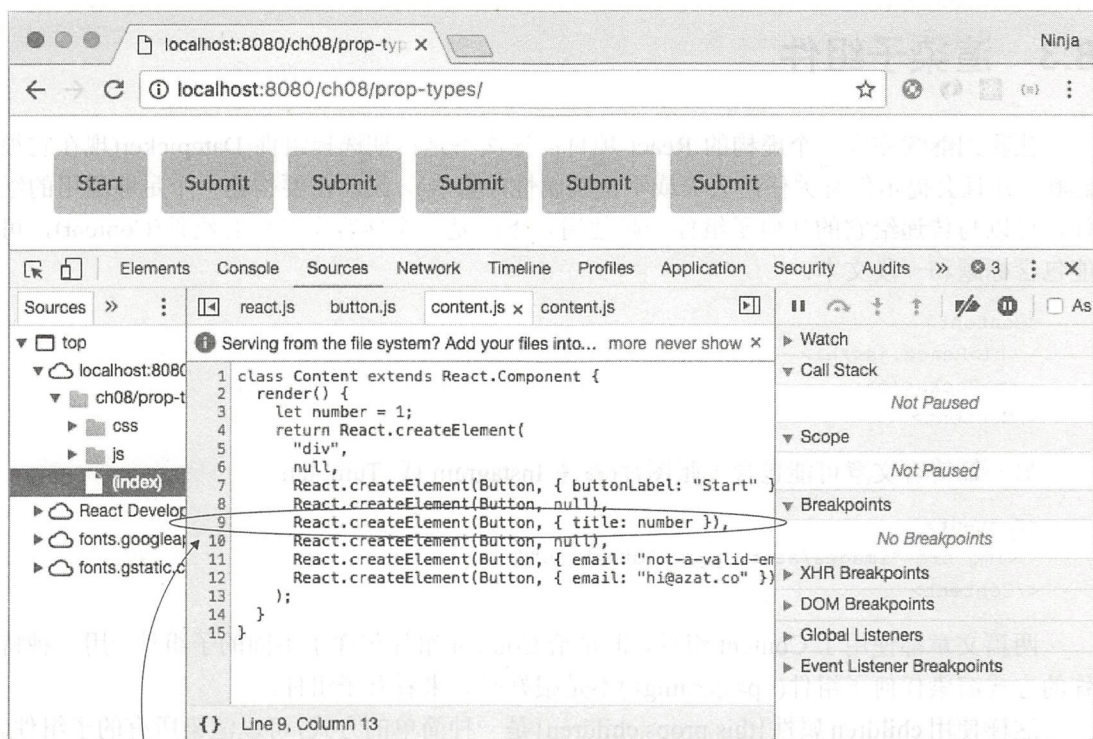
不需要 `source map`(尽管在本书的第 II 部分设置并使用了它们)就知道是第 3 个按钮导致问题的发生。

**注意:** 再重复一次, 只有 React 的未压缩版本(即开发者模式)才会显示这些警告。

尝试使用属性类型检查和验证。这是一个优雅的特性。考虑如下代码, 假设使用了与之前相同的 `Button` 组件:

```
<Button title={number}/>
```

发现问题了吗? 猜测一下会有多少警告(提示: 注意 `handler` 函数和 `title` 属性)?



消息指出content.js中的第9行导致该问题的产生

图 8.4 检查编译的源代码通常足以理解问题

### source map

你将会看到图 8.2 所示的警告，因为 Content 的不严谨写法(这里有意以这种方式写出来，以显示 defaultProps 和 propTypes 的工作原理)。警告标识出问题组件以及组件中发生问题的位置。

但是行号与源代码不匹配，因为它们引用的是编译的 JavaScript 而不是 JSX。要获得正确的行号，需要使用 source-map 插件，如 source-map-support(<https://github.com/evanw/node-source-map-support>)或 Webpack。第 12 章将讨论 Webpack。

也可以采用非 Webpack 的方式来获得 source map 的支持，方法是在 Babel 命令中或在 package.json 的构建脚本中添加 --sourceMaps=true。关于 Babel 的更多选项，请参考 <https://babeljs.io/docs/usage/options/#options>。

在大型项目或开源项目中使用 propTypes(属性类型和自定义验证)很重要。当然，属性类型没有严格模式或错误异常，但好处是，当使用别人的组件时，可以验证提供的属性的类型是否正确。当其他软件工程师使用你的组件时也是如此。他们会感谢你提供对属性类型的正确性验证。这将为每个人带来更好的开发体验！

最后，还有很多其他类型和辅助方法。要查看完整的资料，请参阅 <http://mng.bz/4Lep> 上的文档。



## 8.3 渲染子组件

让我们继续完成一个虚构的 React 项目；这次不是日期选择组件 `Datepicker`(现在它很健壮，并且会提示你有关任何丢失或不正确属性的警告)，因此需要构建一个足够通用的组件，可以与传递给它的任何子组件一起使用。下面是一个博客文章内容组件(`Content`)，可能包含标题和一段文本：

```
<Content>
  <h1>React.js</h1>
  <p>Rocks</p>
</Content>
```

另一篇博客文章可能包含一张图片(参考 `Instagram` 或 `Tumblr`):

```
<Content>
  
</Content>
```

两篇文章都使用了 `Content` 组件，但是给 `Content` 组件传递了不同的子组件。用一种特殊的方式渲染任何子组件(`<p>`或`<img>`)不是很好吗？来看看子组件。

这样使用 `children` 属性 `{this.props.children}` 是一种简单的方式，可以渲染所有的子组件。除了渲染外，也可以做更多的事情。例如，添加`<div>`并顺传子元素：

```
class Content extends React.Component {
  render() {
    return (
      <div className="content">
        {this.props.children}
      </div>
    )
  }
}
```

`Content` 作为父节点，包含`<h1>`和`<p>`子节点：

```
ReactDOM.render(
  <div>
    <Content>
      <h1>React</h1>
      <p>Rocks</p>
    </Content>
  </div>,
  document.getElementById('content')
)
```

最终结果是`<h1>`和`<p>`被`<div>`容器包裹，该容器有 `content` 类，如图 8.5 所示。记住，对于类属性，在 React 中使用 `className`。

显然，可以向 `Content` 组件添加更多的内容；例如，更多的类用于样式、布局，甚至与事件和状态交互的可访问属性。使用 `this.props.children`，可以创建灵活、强大和通用的



透传组件。

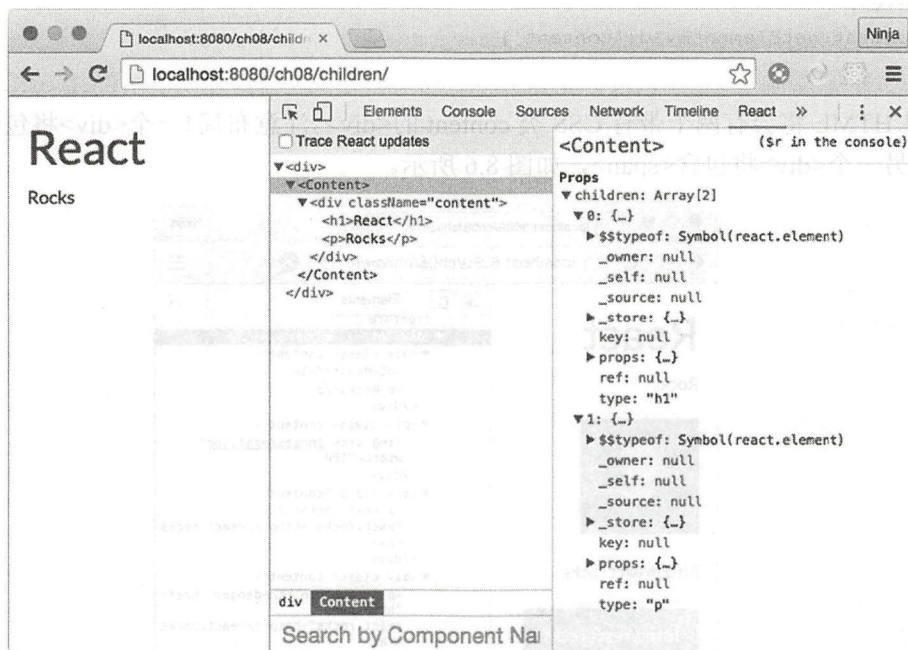


图 8.5 使用 `this.props.children` 呈现带有标题和段落的单个 `Content` 组件，显示其中的两项

假设需要显示除文本和图像之外的链接或按钮，如上例所示。`Content` 组件仍然是类名为 `content` (`className` 属性)的 `<div>` 包装器，但是现在会有更多不同的子组件。好处是 `Content` 组件的子组件可以是未知的<sup>2</sup>。无须改变 `content` 类。

当实例化 `content` 类时，把子组件置于 `Content` 组件中，参见代码清单 8.3(ch08/children/jsx/script.jsx)。

#### 代码清单8.3 使用Content渲染元素

```
ReactDOM.render(
  <div>
    <Content>
      <h1>React</h1>
      <p>Rocks</p>
    </Content>
    <Content>
      
    </Content>
    <Content>
      <a href="http://react.rocks">http://react.rocks</a>
    </Content>
    <Content>
      <a className="btn btn-danger"
        href="http://react.rocks">http://react.rocks</a>
    </Content>
  </div>
);
```

<sup>2</sup> 在 IT 环境中，“未知”指的是泛化的东西，以便可以在各种系统之间协作共用，参见 <http://whatis.techtarget.com/definition/agnostic>

```

    </Content>
  </div>,
  document.getElementById('content')
)

```

结果 HTML 将会有两个带有 CSS 类 `content` 的 `<div>`。注意布局!一个 `<div>` 将包含 `<h1>` 和 `<p>`, 另一个 `<div>` 将包含 `<span>`, 如图 8.6 所示。



图 8.6 在一个组件类中渲染 4 种不同元素

关于 `children` 属性有趣的是, 如果有多个子元素(参见前面的图 8.5), 它可以是一个数组。可以访问以下各个元素:

```

{this.props.children[0]}
{this.props.children[1]}

```

验证子组件时要小心, 当只有一个子元素时, `this.props.children` 不是数组。如果使用了 `this.props.length` 并且唯一的子元素是字符串, 将会导致 bug 产生。因为 `length` 是一个有效的字符串属性。使用 `React.Children.count(this.props.children)` 获取子组件的准确计数。

React 还包含其他辅助方法。最有趣的(在我看来)是下面这些:

- `React.Children.map()`
- `React.Children.forEach()`
- `React.Children.toArray()`

没有理由去重复不断变化的列表, 可以在 <http://mng.bz/Oi2W> 上找到官方文档。

## 8.4 创建 React 高阶组件以实现代码复用

继续假设你为一支大型团队工作, 并创建其他开发人员要在项目中使用的组件。假设你正在处理一个接口, 三位团队成员要求你实现一种加载资源(React 网站)的方式, 但每个

人都希望使用他们自己的可视化形式来表示按钮、图像和链接。也许可以实现一个方法并在事件处理程序中调用它，但是有一个更优雅解决方案：高阶组件。

高阶组件(Higher-Order Component, HOC)可以让你使用额外的逻辑来增强组件(见图 8.7)。当与 HOC 一起使用时，可以将此模式视为组件功能继承。换句话说，HOC 让你能够复用代码。这允许你和你的团队成员在 React 组件之间共享功能。通过这样做，可以避免重复自身(DRY, <http://mng.bz/1K5k>)。

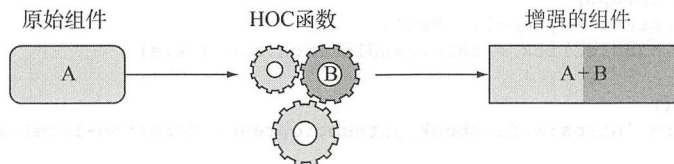


图 8.7 高阶组件模式的简化表示，其中增强的组件不仅具有 A 的属性，也具有 A 和 B 的属性

实际上，HOC 是 React 的组件类，它们渲染原始类，同时在此过程中添加额外的功能。定义 HOC 很简单，因为它只是一个函数。可以用一个箭头声明 HOC：

```
const LoadWebsite = (Component) => {  
  ...  
}
```

名称 LoadWebsite 是任意的，可以将 HOC 命名为任何内容，只要在增强组件时使用相同名称即可。对于函数(LoadWebsite)的参数也是如此。这是原始组件(尚未增强)。

为了演示，让我们为你的三位同事设立一个项目。项目结构如下，在 elements.jsx 中有三个无状态组件——Button、Link 和 Logo，并且 HOC 函数在 load-website.jsx 中：

```
/hi-order  
/css  
  bootstrap.css  
  style.css  
/js  
  content.js  
  elements.js  
  load-website.js  
  react.js  
  react-dom.js  
  script.js  
/jsx  
  content.jsx  
  elements.jsx  
  load-website.jsx  
script.jsx  
index.html  
logo.png
```

你的同事需要一个标签和一个单击事件处理程序。让我们设置标签并定义 handle Click()方法，此挂载事件演示了组件生命周期，参见代码清单 8.4(ch08/hi-order/jsx/load-website.jsx)。



## 代码清单8.4 实现高阶组件

可以是一个字符串常量，因为不需要一个this的实例，但是这种方法能使组件自身保持完善

```
const LoadWebsite = (Component) => {
  class _LoadWebsite extends React.Component {
    constructor(props) {
      super(props)
      this.state = {label: 'Run'}
      this.handleClick = this.handleClick.bind(this)
    }
    getUrl() {
      return 'https://facebook.github.io/react/docs/top-level-api.html'
    }
    handleClick(event) {
      var iframe = document.getElementById('frame').src =
        this.getUrl()
    }
    componentDidMount() {
      console.log(ReactDOM.findDOMNode(this))
    }
    render() {
      console.log(this.state)
      return <Component {...this.state} {...this.props} />
    }
  }
  _LoadWebsite.displayName = 'EnhancedComponent'
  return _LoadWebsite
}
```

确保在这个方法中，this一直是指向当前组件的实例

在一个iframe中加载React网站

使用扩展运算符将state和props作为属性传递

定义要显示的HOC名称

这没什么复杂的，对吧？这里用到了本书前面没有介绍的两个技术——displayName和扩展运算符。现在让我们快速查阅一下它们。

### 8.4.1 使用 displayName: 用以区分父组件与子组件

默认情况下，JSX 使用类名作为实例(元素)的名称。因此，在示例中使用 HOC 创建的元素的名称为\_LoadWebsite。

#### JavaScript 中的下划线

在 JavaScript 中，下划线(\_)是(Lodash 和 Underscore 库使用的)名称的有效字符。另外，变量或方法名以下划线开头，通常意味着是私有属性或方法，不能用作公共接口(例如用于另一个模块、类、对象、功能等)。不建议使用私有 API，因为它们可能会更频繁地更改或包含未知的行为。

名称以下划线开头是一个惯例，意味着不受引擎或平台的限制。这只是 JavaScript 软件工程师使用和认可的常见方式。换句话说，当在名称中使用\_时，方法和变量不会自动变为私有。要使变量或方法为私有，请使用闭包，详情参阅 <http://developer.mozilla.org/en/docs/Web/JavaScript/Closures> 和 <http://javascript.crockford.com/private.html>。

当需要更改此名称时，有名为 displayName 的静态属性可用。你可能知道，ES6 中的

静态类属性必须在类定义之外定义(在撰写本书时,静态属性的标准尚未定稿)。

总而言之,当 React 的元素名和组件类名不同时,需要设置 `displayName`,如图 8.8 所示。可以看到在 `load-website.jsxHOC` 中使用 `displayName` 来增加名称十分有用,因为默认的组件名就是函数名(可能并不总是想要的名称)。

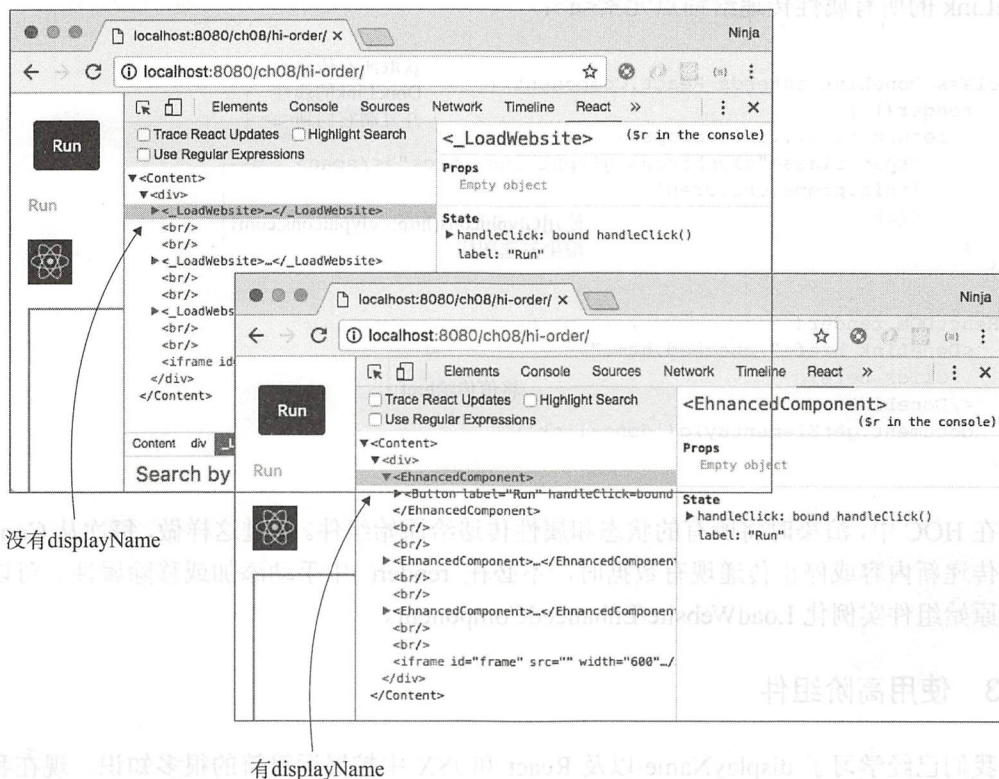


图 8.8 通过使用 `displayName` 这个静态属性,可以将组件的名称从 `_LoadWebsite` 改为 `EnhancedComponent`

### 8.4.2 使用扩展运算符: 传递所有属性

接下来,我们来看看扩展运算符(`...`),它是 ES6+/ES2015+数组的一部分(<http://mng.bz/8fjN>);在撰写本书时,建议在对象上使用扩展运算符(<https://github.com/sebmarkbage/ecmascript-rest-spread>)。React 团队很自然地 JSX 添加了对扩展运算符的支持。

这个概念并不复杂。在元素中使用时,扩展运算符允许将对象(`obj`)的所有属性作为属性传递:

```
<Component {...obj}/>
```

在 `load-website.jsx` 中,在渲染原始组件时,使用扩展运算符传递状态和属性变量到原始组件。这里需要扩展运算符,因为无法提前知道函数的所有属性是否将之作为参数;因此,扩展运算符是概括性声明,用于传递所有数据(在变量或对象中)。

在 React 和 JSX 中,可以使用多个扩展运算符或混合使用扩展运算符和传统的 `key = value` 属性声明。例如,可以将所有状态和属性,也包括 `className`,从当前类传递给新的



元素组件:

```
<Component {...this.state} {...this.props} className="main" />
```

考虑一个含有子组件的例子。在这种情况下, 结合 `this.props` 使用扩展运算符会将 `DoneLink` 的所有属性传递给锚点元素 `<a>`:

```
class DoneLink extends React.Component {
  render() {
    return <a {...this.props}>
      <span class="glyphicons glyphicons-check"></span>
      {this.props.children}
    </a>
  }
}

ReactDOM.render(
  <DoneLink href="/checked.html">
    Click here!
  </DoneLink>,
  document.getElementById('done-link')
)
```

获取所有传递给  
DoneLink 的属性  
并复制它们到 `<a>`

使用 Glyphicons(<http://glyphicons.com>)  
渲染复选图标

将值传给 href

在 HOC 中, 渲染时将所有的状态和属性传递给原始组件。通过这样做, 每次从 `Content` 组件传递新内容或停止传递现有数据时, 不必在 `render()` 中手动添加或移除属性。可以为每个原始组件实例化 `LoadWebsite/EnhancedComponent`。

### 8.4.3 使用高阶组件

我们已经学习了 `displayName` 以及 `React` 和 `JSX` 中扩展运算符的很多知识。现在我们来了解一下如何使用 HOC。

我们回到 `Content` 组件和 `content.jsx`, 那里使用了 `LoadWebsite`。定义 HOC 后, 需要在 `content.jsx` 中创建使用 HOC 的组件:

```
const EnhancedButton = LoadWebsite(Button)
const EnhancedLink = LoadWebsite(Link)
const EnhancedLogo = LoadWebsite(Logo)
```

现在, 可以实现三个组件——`Button`、`Link` 和 `Logo`——以重新使用 HOC 模式的代码。`Button` 组件是通过 `LoadWebsite` 创建的, 所以可以巧妙继承其属性(`this.props.handleClick` 和 `this.props.label`):

```
class Button extends React.Component {
  render() {
    return <button
      className="btn btn-primary"
      onClick={this.props.handleClick}>
      {this.props.label}
    </button>
  }
}
```



Link 组件通过 HOC 创建，因此仍然可以使用 handleClick 和 label 属性：

```
class Link extends React.Component {
  render() {
    return <a onClick={this.props.handleClick} href="#">
      {this.props.label}</a>
    }
  }
}
```

最后，Logo 组件使用同样的属性。猜对了：它们之所以巧妙，是因为当在 content.jsx 中创建 Logo 时，使用了扩展运算符：

```
class Logo extends React.Component {
  render() {
    return 
  }
}
```

这三个组件有不同的渲染，但是它们都可以从 LoadWebsite 获取 this.props.handleClick 和 this.props.label。父组件 Content 渲染的元素如代码清单 8.5 所示(ch08/hi-order/jsx/content.jsx)。

#### 代码清单8.5 HOC共享事件处理程序

```
const EnhancedButton = LoadWebsite(Button)
const EnhancedLink = LoadWebsite(Link)
const EnhancedLogo = LoadWebsite(Logo)

class Content extends React.Component {
  render() {
    return (
      <div>
        <EnhancedButton />
        <br />
        <br />
        <EnhancedLink />
        <br />
        <br />
        <EnhancedLogo />
        <br />
        <br />
        <iframe id="frame" src="" width="600" height="500"/>
      </div>
    )
  }
}
```

声明一个iframe，click()方法将React网站加载到其中

最后，不要忘记在 script.jsx 的最后渲染 Content：

```
ReactDOM.render(
  <Content />,
  document.getElementById('content')
)
```

当打开此页面时,上面含有三个元素(Button、Link 和 Logo)。这些元素都有相同的功能:当被单击时,加载 iframe,如图 8.9 所示。

① 点击这三个元素中的任何一个

② 加载网站

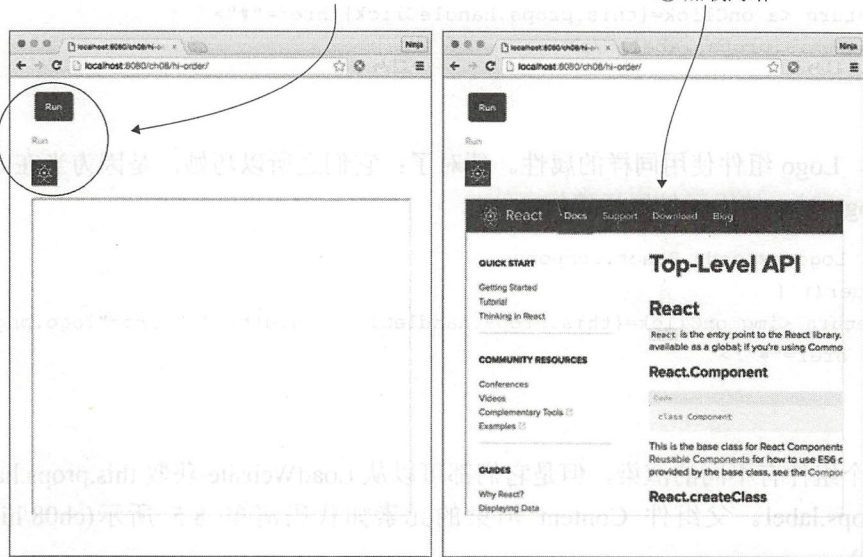


图 8.9 所有三个组件都加载 React 网站,感谢提供加载代码的函数

如你所见,HOC 非常适合抽象代码。可以用来编写自己的迷你模块,这些模块是可复用的 React 组件。HOC 以及属性类型是创建利于开发的组件的工具,其他人也将会喜欢使用。

## 8.5 最佳实践: 展示组件与容器组件

有个特性允许根据代码和团队规模扩展 React 代码:展示组件与容器组件。我们已经在前几章接触过它们,但是现在,由于知道了传递子组件和 HOC,因此很容易推出容器组件。

一般来说,将代码分为两种类型可以使其更简单,更易于维护。展示组件通常只添加 DOM 结构和样式。它们有属性,但是往往没有状态。大多数情况下,可以使用无状态展示组件的功能。例如,Logo 组件是展示组件类风格的很好例证:

```
class Logo extends React.Component {
  render() {
    return 
  }
}
```

或用于表现函数式风格:

```
const Logo = (props)=>{
  return   
  }
```

展示组件通常使用 `this.props.children` 作为包装器来创建子组件，例子包括 `Button`、`Content`、`Layout`、`Post` 等。但它们很少处理数据或状态，这是容器组件要做的工作。

容器组件通常由 HOC 生成以注入数据源。它们有状态，示例包括 `SaveButton`、`ImagePostContent` 等。理论上，展示组件和容器组件都可以包含其他展示组件或容器组件，但在实践中，通常会使展示组件仅包含其他展示组件，使容器组件包含其他容器组件或展示组件。

最好的方法是从满足需求的组件开始。如果开始看到重复的模式，或看到通过多层嵌套组件传递的属性又不在中间组件中使用，请引入容器组件或两者都引入。

注意：你可能会听到诸如木偶组件或瘦组件，以及智能组件或胖组件这样的术语。这些是展示组件和容器组件的同义词，后者是最新添加的 React 术语。

## 8.6 测验

1. React 提供了强大的验证功能，在服务器端无须检查输入。这么理解正确还是错误？
2. 除了使用 `defaultProps` 设置属性以外，还可以在构造函数中使用 `this.props.NAME = VALUE` 来设置。这么理解正确还是错误？
3. `children` 属性可以是数组或节点。这么理解正确还是错误？
4. 高阶组件模式可以通过函数来实现。这么理解正确还是错误？
5. 压缩的开发版和未压缩的生产版 React 库文件，主要区别是压缩版有警告，未压缩版是优化过的代码。这么理解正确还是错误？

## 8.7 小结

- 通过设置组件的 `defaultProps` 属性，可以给任意组件属性设置默认值。
- 当使用未压缩的开发版 React 库时，可以增强对组件属性值的验证。
- 如果需要，可以验证属性类型，设置为 `isRequired` 来强制使用或定义自定义验证。
- 如果属性值没有通过验证，会在浏览器的控制台中出现一条警告。
- React 库的压缩生产版不包括这些验证检查。
- React 允许在组件中，通过高阶组件封装和复用普通的属性、方法和事件。
- 高阶组件的定义类似函数，使用另一个组件作为参数，此参数由该组件继承于 HOC。
- 任何嵌套在 JSX 元素中的 HTML 或 React 组件，都可以通过父组件的 `props.children` 属性来访问。



## 8.8 测验答案

1. 错误。前端验证不能替代服务器端验证。前端代码对任何人都是暴露的，任何人都可以通过逆向前端应用来绕过它与服务器进行通信，并将任意数据直接发送到服务器。
2. 错误。当创建元素时，React 需要 `defaultProps` 作为静态类属性，但 `this.props` 是实例属性。
3. 正确。如果只有一个子组件，那么 `this.props.children` 是单节点。
4. 正确。HOC 模式被实现为一个函数，它接收一个组件并创建另一个具有增强功能的组件类。这个新类在渲染时将属性和状态传递给原始组件。
5. 正确。压缩版不会显示警告。

# 第9章

## 项目：菜单组件

### 本章内容：

- 理解项目结构和脚手架(scaffolding)
- 不使用 JSX 构建菜单组件
- 使用 JSX 构建菜单组件

接下来的三章将介绍几个项目，这几个项目将以你在第 1 至第 8 章所学的知识为基础。通过重复 React 中最重要的一些技术和想法，这些项目将增强这些知识。第一个项目最小，但不要跳过它。

假设你正基于一个统一的可视化框架开展工作，将在公司的所有应用中使用这个框架。在多种应用中具有相同的观感很重要。考虑一下 Twitter Bootstrap 是如何在许多 Twitter 应用中使用的，以及谷歌的 Material UI<sup>1</sup>如何应用于谷歌的如下许多特性：AdWords、Google Analytics、搜索、驱动器、文档等。

第一个任务是实现图 9.1 所示的菜单，它被用在各种应用程序中许多页面

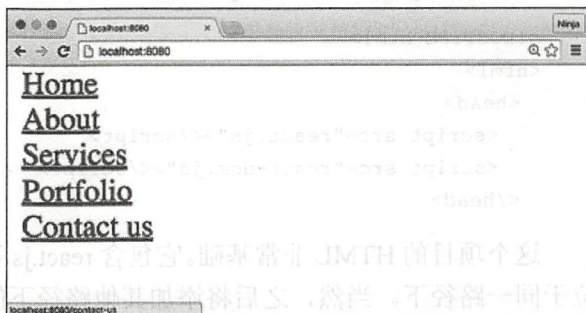


图 9.1 实现了将要构建的菜单

<sup>1</sup> Twitter Bootstrap: <http://getbootstrap.com>。实现了 Twitter Bootstrap 的 React 组件: <https://react-bootstrap.github.io>。谷歌 Material Design: <https://material.io>。实现了 Material Design 的 React 组件: [www.material-ui.com](http://www.material-ui.com)

的布局标题中。菜单项需要根据用户角色和当前正在查看的应用程序页面进行更改。例如，管理员和管理者应该看到用于管理用户菜单的选项。同时，此布局将会用于那些需要独立菜单选项集的客户关系应用。现在有了思路，菜单应该动态生成，这意味着需要一些生成菜单选项的 React 代码。

为简单起见，菜单项将只是标签。将创建两个自定义的 React 组件：Menu 和 Link。创建方式与创建第 1 章中的 HelloWorld 组件相似——可以使用此方式创建任何组件。

该项目将展示如何以编程方式渲染嵌套元素。手动编写固定菜单项不是一种好的方式。当需要更改项目时会发生什么？它不是动态的！所以将使用 `map()` 函数来执行此操作。

注意：为跟踪此项目，需要下载未压缩版本的 React(这样当出现问题时，会返回有用的警告)。也可以下载并安装 Node.js 和 npm。它们对于这个项目而言不是绝对必要的，但在本章后面编写 JSX 时很有用。附录 A 中涵盖了这两种工具的安装方法。

## 9.1 项目结构和脚手架

让我们从综述项目结构开始吧。为了保持简单，项目结构是扁平化的：

```
/menu
  index.html      <—— 主要的HTML文件
  package.json
  react-dom.js
  react.js
  script.js       <—— 主要的script文件
```

记住，这些是在此项工作结束之后才有的。应该从一个空文件夹开始。所以，下面创建一个新的文件夹并开始实现项目：

```
$ mkdir menu
$ cd menu
```

下载 `react.js` 和 `react-dom.js` 的版本 15，将它们放置在文件夹中，以下是 HTML 文件：

```
<!DOCTYPE html>
<html>
  <head>
    <script src="react.js"></script>
    <script src="react-dom.js"></script>
  </head>
```

这个项目的 HTML 非常基础。它包含 `react.js` 和 `react-dom.js` 文件，这两个文件和 HTML 位于同一路径下。当然，之后将添加其他路径下的自定义\*.js 文件，如 `js` 或 `src`。

`<body>` 只包含两个元素。一个元素是 `<div>` 容器，id 是 `menu`；这是菜单将要被渲染的位置。另一个元素是 `<script>` 标签，包含 React 应用代码：

```
<body>
  <div id="menu"></div>
```



```
<script src="script.js"></script>
</body>
</html>
```

至此，已经完成了脚手架。这是不使用 JSX 时构建菜单的基础。

## 9.2 不使用 JSX 构建菜单

script.js 是应用的主文件。它包含 ReactDOM.render() 以及两个组件，参见代码清单 9.1(ch09/menu/ script.js)。

代码清单9.1 菜单脚本的基本框架

```
class Menu extends React.Component {...}           ← 定义菜单

class Link extends React.Component {...}             ← 定义菜单使用的链接

ReactDOM.render(
  React.createElement(
    Menu,
    null,
  ),
  document.getElementById('menu')
)
```

← 不要给菜单传递任何属性

当然，可以使菜单依赖外部的菜单项列表，菜单项列表可以是在其他地方定义的诸如 menuOptions 这样的属性提供的：

```
const menuOptions = [...]
//...

ReactDOM.render(
  React.createElement(
    Menu,
    {menus: menuOptions}
  ),
  document.getElementById('menu')
)
```

这两种方法都有效，可以根据对这个问题的回答来选择一个：是需要菜单仅仅有结构和样式，还是要同时获取信息？本章我们将采用后一种方法并使菜单是自身完善的。

### 9.2.1 Menu 组件

现在来创建 Menu 组件。我们来逐步了解代码。要创建 Menu 组件，需要扩展 React.Component()：

```
class Menu extends React.Component {...}
```

Menu 组件将渲染单独的菜单项，即 link 标签。在可以渲染它们之前，需要定义菜单

项。它们被硬编码在 `menus` 数组中(可以从数据模型和存储中获取, 或者在更复杂的场景中从服务器获取), 如下所示:

```
render() {  
  let menus = ['Home', 'About', 'Services', 'Portfolio', 'Contact us']  
  //...  
}
```

← 模拟数据仓库

接下来, 将会返回菜单的 `Link` 元素(它们有四个)。回想一下, `return` 只能有一个元素。因此, 可以使用 `<div>` 封装四个链接, 这是没有属性的用于封装的 `<div>` 元素的开头:

```
return React.createElement('div',  
  null,  
  //... we will render links later
```

值得一提的是, `{}` 不仅可以输出变量或表达式, 还可以输出数组。当有项的列表需要展示时, 这将派上用场。基本上, 要渲染数组的每个元素, 可以将数组传递给 `{}`。虽然 `JSX` 和 `React` 可以输出数组, 但它们不输出对象。因此, 对象必须转换为数组。

知道了可以输出数组, 下面继续生成一个包含 `React` 元素的数组。`map()` 函数很好用, 因为它返回一个数组。可以实现 `map()`, 以便每个元素都是 `React.createElement(Link, {label: v})` 表达式的结果, 并被 `<div>` 封装。在此表达式中, `v` 是 `menus` 数组项(`Home`、`About`、`Services` 等)的值, `i` 是其索引号(0、1、2、3 等):

```
menus.map((v, i) => {  
  return React.createElement('div',  
    {key: i},  
    React.createElement(Link, {label: v})  
  })  
})
```

注意到 `key` 属性被设置为索引 `i` 了吗? 这是必要的, 以便 `React` 可以更快地访问列表中的每个 `<div>` 元素。如果没有设置 `key` 属性, 将会看到以下警告(至少在 `React` 版本 15、0.14 和 0.13 中如此):

```
Warning: Each child in an array or iterator should have a unique "key" prop.  
Check the render method of `Menu`. See https://fb.me/react-warning-keys for  
more information.
```

```
in div (created by Menu)  
in Menu
```

再次称赞 `React` 良好的错误和警告信息。

因此, 列表中的每个元素都必须具有唯一的 `key` 属性值。它们不一定在整个应用中唯一或跨组件唯一, 只是在此列表中唯一。有趣的是, 从 `React v15` 开始, 在 `HTML` 中看不

到 key 属性(这是一件好事,可以避免污染 HTML)。但是 React DevTools 显示了 key,如图 9.2 所示。

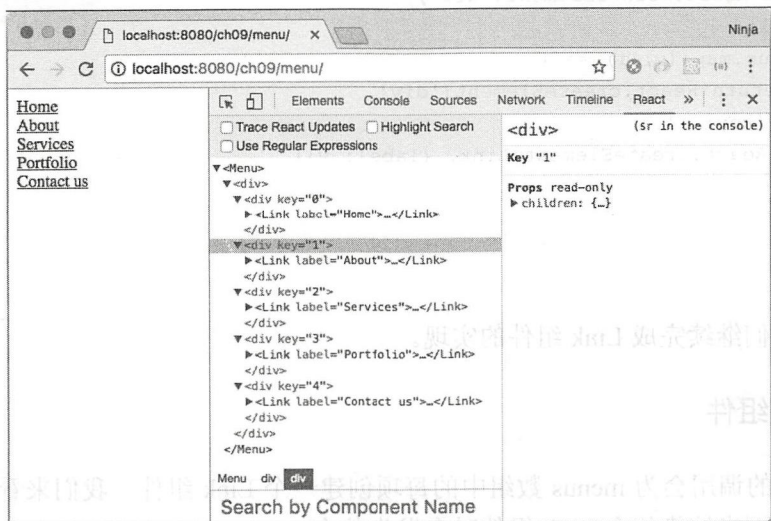


图 9.2 React DevTools 显示了列表元素的 key

### Array.map() 函数

经常在 React 组件中使用 Array 类的 map() 函数以表示数据列表。这是因为当创建 UI 时,要使用数组表示的数据。UI 也是数组,但元素稍有不同(React 元素! )。

map() 在数组上被调用,它返回从原始数组元素通过一个函数转换成的新数组元素。至少在使用 map() 时,需要传递此函数:

```
[1, 2, 3].map( value => <p>value</p>)
// <p>1</p><p>2</p><p>3</p>
```

除了项值(value)之外,还可以使用额外的两个参数——index 和 list:

```
[1, 2, 3].map( (value, index, list) => {
  return <p id={index}>{list[index]}</p>
}) // <p id="0">1</p><p id="1">2</p><p id="2">3</p>
```

此 <div> 有 key 属性,这很重要。它允许 React 通过将列表转换为散列来优化列表的渲染,并且哈希的访问时间优于列表或数组。基本上,在数组中创建了众多的 Link 组件,并且每个 Link 组件都使用 menus 数组中的 label 属性值。

代码清单 9.2 中是完整的 Menu 代码(ch09/menu/script.js), 十分简单直接。

### 代码清单9.2 Menu组件使用map()渲染链接

```
class Menu extends React.Component {
  render() {
    let menus = ['Home',
      'About',
      'Services',
```



```
    'Portfolio',  
    'Contact us']  
    return React.createElement('div',  
      null,  
      menus.map((v, i) => {  
        return React.createElement('div',  
          {key: i},  
          React.createElement(Link, {label: v})  
        )  
      })  
    )  
  })  
}
```

现在，我们继续完成 Link 组件的实现。

## 9.2.2 Link 组件

对 `map()` 的调用会为 `menus` 数组中的每项创建一个 Link 组件。我们来看看 Link 组件的代码，看一下当渲染每个 Link 组件时会发生什么。

在 Link 组件的渲染代码中，可以编写一个表达式来创建一个 URL，该 URL 将用于 `<a>` 标签的 `href` 属性。当创建 Link 时，`this.props.label` 的值将从 Menu 传递到 Link。在 Menu 组件的 `render()` 函数中，Link 元素在 `map` 的闭包/迭代器函数中创建(使用 `React.createElement(Link, {label: v})`)。

`label` 属性用于构建 URL 拼接(必须小写，不含空格)：

```
class Link extends React.Component {  
  render() {  
    const url = '/'  
      + this.props.label  
      .toLowerCase()  
      .trim()  
      .replace(' ', '-')  
  }  
}
```

方法 `toLowerCase()`、`trim()` 和 `replace()` 是标准的 JavaScript 字符串函数。它们执行的操作分别是：转换为小写、裁剪边缘空白和用虚线替换空白空格。

URL 表达式会生成如下 URL：

- `/home` 对应 Home
- `/about` 对应 About
- `/services` 对应 Services
- `/portfolio` 对应 Portfolio
- `/contact-us` 对应 Contact US

现在可以实现 Link 的 UI: `render()` 的返回值。在 `render()` 函数返回的 Link 组件中，将 `this.props.label` 作为第三个参数传递给 `createElement()`。它将成为 `<a>` 标签内容(链接文本)的一部分。Link 可以渲染此元素：

```
//...
return React.createElement(
  'a',
  {href: url},
  this.props.label
)
}
}
```

但最好使用换行符(<br>)分隔每个链接。并且由于组件必须返回单个元素，需要在一个 Div 容器(<div>)中封装锚点元素(<a>)和换行符(<br>)。因此，在 Link 组件的 render() 中返回不包含属性的 div：

```
//...
return React.createElement('div',
  null,
  //...
```

createElement() 函数的第 2 个参数之后的每个参数(例如第 3、第 4 和第 5 个参数)将作为内容(子组件)。要创建链接元素，将其作为第 3 个参数传递。并且要在每个链接之后创建一个换行元素，将换行元素<br>作为第 4 个参数传递。

```
//...
return React.createElement('div',
  null,
  React.createElement(
    'a',
    {href: url},
    this.props.label
  ),
  React.createElement('br')
)
}
})
```

代码清单 9.3 中是 Link 组件的完整代码(ch09/menu/script.js)。url 函数可以作为类方法或组件外部方法来创建。

代码清单9.3 Link组件

```
class Link extends React.Component {
  render() {
    const url = '/'
      + this.props.label
        .toLowerCase()
        .trim()
        .replace(' ', '-')
    return React.createElement('div',
      null,
      React.createElement(
        'a',
        {href: url},
        this.props.label
```

定义一个函数，在菜单  
名称外创建URL片段

将URL片段传  
递给href属性

```
    ),  
    React.createElement('br')  
  )  
}  
}
```

给每个菜单项添加一个换行元素

下面运行菜单组件。

### 9.2.3 运行菜单组件

为了看到图 9.3 所示页面，使用 Chrome、Firefox、Safari 或 Internet Explorer 浏览器打开此文件。这样就可以了。这个项目不需要任何编译。

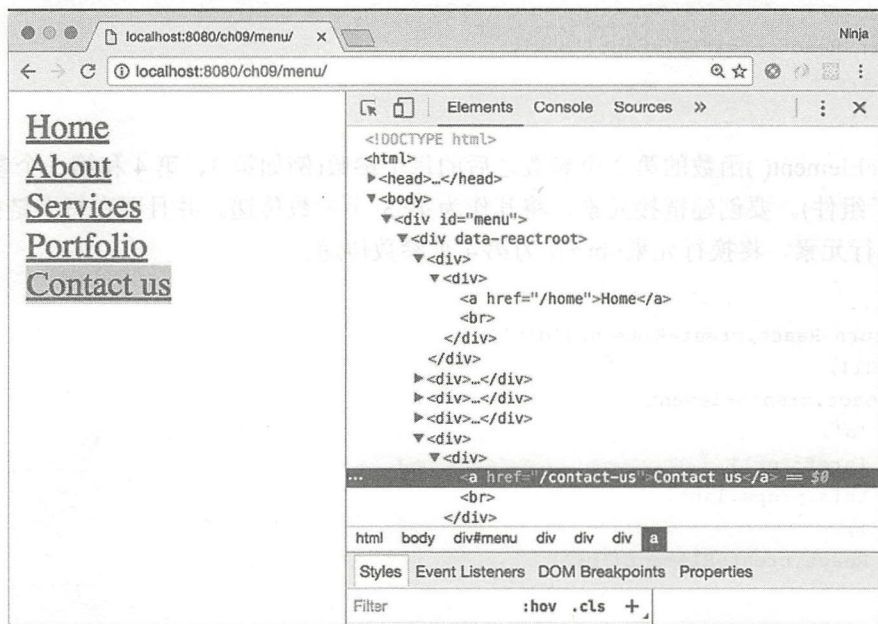


图 9.3 React 菜单展示嵌套组件的渲染

#### 使用本地 Web 服务器

当打开示例页面时，地址栏中显示的协议是 file:///...，这不够理想，但是对于这个项目可用。在真实的开发环境中，需要一台 Web 服务器；使用 Web 服务器，协议是 http:///... 或 https:///...，如图 9.3 所示。

是的，即便是简单的 Web 页面(就像这个示例)，我也更喜欢使用一台本地 Web 服务器。它使运行代码更接近于生产中的样子。此外，也可以使用 AJAX/XHR，但如果是在浏览器中打开本地 HTML 文件，则无法使用。

运行本地 Web 服务器的最简单方法是使用 node-static(www.npmjs.com/package/node-static)或类似的 Node.js 工具，例如 http-server(www.npmjs.com/package/http-server)。即便在 Windows 系统上也是如此，虽然我已经很多年没有使用 Windows 系统了。如果不喜欢使用 Node.js，那么替代方案包括 IIS、Apache HTTP 服务器、NGINX、MAMP、LAMP 以及 Web 服务器的其他变种。不作过多赘述，Node.js 工具因其极简、轻量级特性被强烈



推荐使用。

使用 npm 安装 node-static:

```
$ npm install -g node-static@0.7.6
```

安装完毕后, 从项目的根文件夹(或父文件夹)运行此命令, 使通过 `http://localhost:8080` 可访问到该文件。这不是一个外部链接——在单击这个链接之前, 运行下面的命令:

```
$ static
```

如果在 `react-quickly/ch09/menu` 路径下运行 `static` 命令, 那么 URL 会是 `http://localhost:8080`。相反, 如果在 `react-quickly` 路径下运行 `static`, 那么 URL 会是 `http://localhost:8080/ch09/menu`。

在 Mac OS 或 UNIX/Linux(POSIX 系统)下可以按 `Ctrl+C` 组合键来终止服务器。对于在 Windows 系统下是否也可以, 尚不确定!

毫无疑问, 页面应该显示 5 个链接(或者更多个链接, 如果向 `menus` 数组中添加了菜单项的话), 如前图 9.1 所示。这比复制和粘贴 5 个 `<a>` 元素, 并且在粘贴完之后在多处修改标签和 URL 更好。并且如果使用 JSX, 该项目会变得更好。

## 9.3 在 JSX 中构建菜单

将要构建的这个项目会更广泛, 包括 `node_modules`、`package.json` 和 JSX:

```
/menu-jjsx
/node_modules
index.html
package.json
react-dom.js
react.js
script.js
script.jsx
```

JSX转JS的Babel开发依赖

主JSX程序

如你所见, 有一个用于开发依赖的 `node_modules` 文件夹, 例如用于 JSX 转 JS 的 Babel。

**注意:** 虽然可以将 `react` 和 `react-dom` 作为 npm 模块安装而不是作为文件引用, 但是如果这样部署, 会导致额外的复杂性。现在, 要部署此应用, 只需要复制不包括 `node_modules` 的项目文件夹。如果使用 npm 安装 React 和 ReactDOM, 同样也要包含这个文件夹, 使用打包器(bundler)或将 `dist` 中的 JS 文件复制到根路径中(它们已经在这里了)。所以, 在此例中, 我们在根路径下使用文件。本书在第 2 章中包含了打包器, 但是现在让我们将事情保持简单。

创建一个新文件夹:

```
$ mkdir menu-jjsx
$ cd menu-jjsx
```

然后, 使用 `npm init -y` 命令创建 `package.json` 文件。将代码清单 9.4 中的代码添加到

package.json 文件中，以此来安装和配置 Babel(ch09/menu-jsx/package.json)

代码清单9.4 在JSX中构建Menu组件的package.json

```
{
  "name": "menu-jsx",
  "version": "1.0.0",
  "description": "",
  "main": "script.js",
  "scripts": {
    "build": "./node_modules/.bin/babel script.jsx -o script.js -w"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": ["react"]
  },
  "devDependencies": {
    "babel-cli": "6.9.0",
    "babel-preset-react": "6.5.0"
  }
}
```

定义构建脚本，并使用-w参数

配置Babel来转换React的JSX语法

包含Babel CLI并将React/JSX加入预设

使用 `npm i` 或 `npm install` 安装开发依赖包。现在已经完成配置。

观察 `script.jsx`，在较高的层次上，它需要包含如下部分：

```
class Menu extends React.Component {
  render() {
    //...
  }
}

class Link extends React.Component {
  render() {
    //...
  }
}

ReactDOM.render(<Menu />, document.getElementById('menu'))
```

看起来很熟悉，对吧？看起来和不使用 JSX 语法有相同的结构。在这个较高的层次上，主要的改变是——`ReactDOM.render()` 中的 `Menu` 组件使用下面这行代码替换了 `createElement()`：

```
ReactDOM.render(<Menu />, document.getElementById('menu'))
```

接下来，将要重构 `Menu` 组件。

### 9.3.1 重构 Menu 组件

`Menu` 组件的开始部分是相同的：

```
class Menu extends React.Component {
  render() {
    let menus = ['Home',
```

```

    'About',
    'Services',
    'Portfolio',
    'Contact us']
  return //...
}
}

```

在 Menu 组件的重构示例中，需要输出值 `v` 作为标签的属性值(即 `label={v}`)。换言之，将 `v` 的值作为属性分配给 `label`。所以，创建 Link 元素的代码从

```
React.createElement(Link, {label: v})
```

变为 JSX 代码：

```
<Link label={v}/>
```

第二个参数(`{label: v}`)的 `label` 属性变为属性 `label={v}`。属性值 `v` 使用 `{}` 声明，因此变为动态的(与固定编码值相比较)。

**注意：**当使用花括号分配属性值时，不需要双引号(“ ”)。

React 还需要 `key={i}` 属性来使列表更高效。因此，最终使用 JSX 语法重构的 Menu 组件参见代码清单 9.5(ch09/menu-jsx/script.jsx)。

#### 代码清单9.5 使用JSX的Menu组件

```

class Menu extends React.Component {
  render() {
    let menus = ['Home',
      'About',
      'Services',
      'Portfolio',
      'Contact us']
    return <div>
      {menus.map((v, i) => {
        return <div key={i}><Link label={v}/></div>
      })}
    </div>
  }
}

```

看到可读性增强了吗？是的！

在 Menu 组件的 `render()` 中，如果希望 `<div>` 在新的一行中，可以使用 `()` 包含它。例如，下面这段代码与代码清单 9.5 完全相同，但是 `<div>` 开始于新的一行，这样可能更具视觉效果：

```

//...
return (
  <div>
    {menus.map((v, i) => {
      return <div key={i}><Link label={v}/></div>
    })}
  </div>
)

```



```
    }}  
  </div>  
)  
}})
```

### 9.3.2 重构 Link 组件

Link 组件中的和<br>标签也需要重构，从

```
//...  
return React.createElement('div',  
  null,  
  React.createElement(  
    'a',  
    {href: url},  
    this.props.label),  
  React.createElement('br')  
)  
//...
```

转换为 JSX 代码：

```
//...  
return <div>  
  <a href={url}>  
    {this.props.label}  
  </a>  
  <br/>  
</div>  
//...
```

Link 组件的完整 JSX 版本参见代码清单 9.6(cho9/menu-jsx/script.jsx)。

#### 代码清单9.6 JSX版本的Link组件

```
class Link extends React.Component {  
  render() {  
    const url = '/'  
      + this.props.label  
        .toLowerCase()  
        .trim()  
        .replace(' ', '-')  
    return <div>  
      <a href={url}>  
        {this.props.label}  
      </a>  
      <br/>  
    </div>  
  }  
}
```

完成了！现在来运行 JSX 项目。

### 9.3.3 运行 JSX 项目

打开命令终端、iTerm 或命令提示符程序。在项目的文件夹(ch09/menu-jsx 或任何其他在下载源代码时命名的文件夹)下,使用 `npm i`(`npm install` 的简写)根据 `package.json` 中的入口安装依赖。

然后,使用 `npm run build` 运行 `npm` 构建脚本。`npm` 脚本会启动带监控标记(`-w` 参数)的 `Babel` 命令,这会保持 `Webpack` 运行,以便可以监控任何文件的更改,并且如果 `JSX` 源代码发生更改,可以将代码从 `JSX` 编译为 `JS`。

无须赘述,监控模式节省了时间。因为每次对源代码修改时,都不需要重新编译。热更新模块对开发更友好,将会在第 12 章中介绍它。

构建脚本中的实际命令如下(但是谁会想用?它太长了!):

```
./node_modules/.bin/babel script.jsx -o script.js -w
```

如果需要进一步了解 `Babel CLI`,请参阅第 3 章。可以在那里找到所有的细节。

在我的电脑上,`Babel CLI` 显示的信息是(在不同的电脑上,路径可能不同):

```
> menu-jsx@1.0.0 build /Users/azat/Documents/Code/react-quickly/ch09/menu-jsx
> babel script.jsx -o script.js -w
```

一切准备就绪,随着 `script.js` 的生成,可以使用 `static`(`node-static` 在 `npm` 上是: `npm i -g node-static`)命令来启动本地主机上的 `HTTP` 服务。应用无论是从外观还是从工作方式上都与常规的 `JavaScript` 语言相同,如图 9.4 所示。

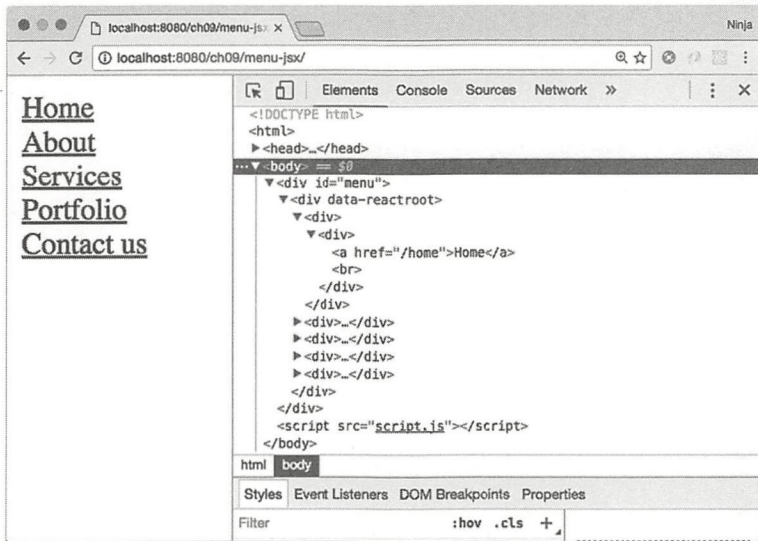


图 9.4 使用 JSX 创建的菜单

## 9.4 测验

为了获得更多收获,不妨尝试下面的操作:

- 通过 Fetch API 从 `menus.json` 文件中加载菜单。关于如何加载数据的相关内容，请参见第 5 章。
- 创建一个 `npm` 脚本，该脚本将从安装到 `node_modules` 中的 `react` `npm` 包提取 `react.js`，并将其复制到 `index.html` 要使用的项目路径下。这将取代在未来的版本中需要手动下载 `react.js` 的方式；改为执行 `npm i react`，然后运行你的脚本。

提交代码到目录 `ch09` 下的一个新的文件夹中，作为对本书的 GitHub 仓库(<https://github.com/azat-co/react-quickly>)的合并请求。

## 9.5 小结

- `key` 属性非常有用，当需要生成列表时设置这个属性。
- `map()` 函数是一种基于原始数组创建新数组的优雅方式，它的迭代参数分别是 `value`、`index` 和 `list`。
- 要使 JSX 正常工作，至少需要 Babel CLI 并预设 React。



# 第 10 章

## 项目：Tooltip 组件

本章内容：

- 理解项目结构和脚手架
- 构建 Tooltip 组件

当浏览包含很多文本的网站时，比如维基百科，我们希望用户获得额外的信息而不丢失自己的位置和上下文语境，从而得到良好的体验。例如，当鼠标悬停在对象上时，可以显示额外的提示(见图 10.1)。这种提示框被称为 tooltip。

React 致力于提供更好的 UI 界面和用户体验，所以非常适合实现 tooltip。让我们构建一个组件，在鼠标悬停时显示帮助文本。

有几个开箱即用的提示工具解决方案，其中包括 `react-tooltip`([www.npmjs.com/package/react-tooltip](http://www.npmjs.com/package/react-tooltip))，但现在的目标是了解 React。从头开始构建 tooltip 是很好的练习。也许你会在日常工作中使用这个例子，把它作为应用程序的一部分，或者将它扩展到开源 React 组件中！

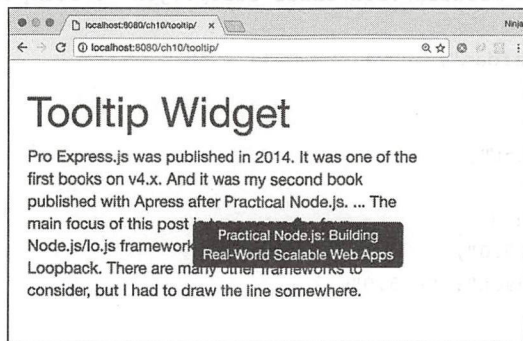


图 10.1 当用户将鼠标移动到标记的文本上时，提示框显示了出来

创建 Tooltip 组件的关键是可以使用任何文本，将它们用 CSS 隐藏起来，当鼠标悬停时显示。你将使用 if/else 条件语句、JSX 和其他编程元素。对于 CSS 部分，将会使用 Twitter Bootstrap 类和特殊主题，使提示工具在短时间内可以达到想要的效果。

注意：为了跟进这个项目，需要下载未压缩的 React 版本，并安装 Node.js 和 npm 来编译 JSX。在这个例子中，将使用 Bootswatch(<https://bootswatch.com/flatly>)中名为 Flatly 的主题。这个主题依赖 Twitter Bootstrap。附录 A 介绍了如何安装它们。

## 10.1 项目结构和脚手架

Tooltip 组件的项目结构如下所示：

```
/tooltip
├── /node_modules
├── bootstrap.css
├── index.html
├── package.json
├── react-dom.js
├── react.js
├── script.js
└── script.jsx
```

← Babel开发依赖，用于将 JSX转译为JS

← 主JSX脚本

类似于第 9 章，有一个 `node_modules` 文件夹用来存放开发依赖，诸如用来转换 JSX 为 JS 的 Babel。项目结构是扁平化的，样式和脚本都在同一个文件夹内。这样做是为了保持简单。当然，在真实的应用中，会将样式和脚本放在不同的文件夹里。

`package.json` 中的关键部分是 npm 构建脚本、Babel 配置、依赖和其他元数据，参见代码清单 10.1。

代码清单10.1 Tooltip项目的package.json文件

```
{
  "name": "tooltip",
  "version": "1.0.0",
  "description": "",
  "main": "script.js",
  "scripts": {
    "build": "./node_modules/.bin/babel script.jsx -o script.js -w"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": ["react"]
  },
  "devDependencies": {
    "babel-cli": "6.9.0",
    "babel-preset-react": "6.5.0"
  }
}
```

在创建了 `package.json` 后，一定要运行 `npm i` 或 `npm install`。

下面开始创建 HTML 部分。创建 `index.html`，如代码清单 10.2 所示(ch10/tooltip/index.html)。

代码清单10.2 Tooltip项目的index.html文件

```
<!DOCTYPE html>
<html>

  <head>
    <script src="react.js"></script>
    <script src="react-dom.js"></script>
    <link href="bootstrap.css"
      rel="stylesheet"
      type="text/css"/>
  </head>

  <body class="container">
    <h1>Tooltip Widget</h1>
    <div id="tooltip"></div>
    <script src="script.js" type="text/javascript"></script>
  </body>
</html>
```

应用样式

定义React和Tooltip的渲染元素

在`<head>`标签中，包含了 React、React DOM 文件和 Twitter Bootstrap 样式。body 元素的内容被最小化：只包含 id 为 `tooltip` 的`<div>`和应用的脚本文件。

接下来，将要创建 `script.jsx`。是的，这并非排版错误。源代码位于 `script.jsx`，但在 HTML 中包含 `script.js` 文件，这是因为将要使用 Babel 的命令行工具。

## 10.2 Tooltip 组件

阅读 `script.jsx`(ch10/tooltip/script.jsx)，里面几乎只有组件的代码和想要渲染的提示文本。提示文本是一个在 `ReactDOM.render()` 中创建 Tooltip 组件时设置的属性，参见代码清单 10.3。

代码清单10.3 Tooltip组件和文本

```
class Tooltip extends React.Component {
  constructor(props) {
    ...
  }
  toggle() {
    ...
  }
  render() {
    ...
  }
}

ReactDOM.render(<div>
  <Tooltip text="The book you're reading now">React Quickly</Tooltip>
  was published in 2017. It's awesome!
</div>,
document.getElementById('tooltip'))
```

声明一个方法来  
显示和隐藏文本

声明必需的  
render()方法

帮助文本作为属性提供，当用户在  
高亮显示的文本上悬停鼠标时，作  
为显示的内容



实现并使用初始属性状态(`opacity:false`)来声明 `Tooltip` 组件。状态(第 4 章有关于状态的更详细描述)控制着帮助文本的显示和隐藏。下面是使用的 `constructor()` 方法:

```
class Tooltip extends React.Component {
  constructor(props) {
    super(props)
    this.state = {opacity: false}
    this.toggle = this.toggle.bind(this)
  }
  ...
}
```

初始状态下隐藏了帮助文本。切换时会更改状态和提示文本的可视性,即是否显示帮助文本。下面来实现 `toggle()` 函数。

### 10.2.1 `toggle()` 函数

现在将定义 `toggle()` 函数,通过将 `opacity` 属性设置为与之前相反的方式来切换提示文本的可视性(`true` 变为 `false`, 或 `false` 变为 `true`):

```
toggle() {
  const tooltipNode = ReactDOM.findDOMNode(this)
  this.setState({
    opacity: !this.state.opacity,
    ...
  })
}
```

可以使用在第 4 章学到的 `this.setState()` 方法来改变 `opacity` 属性。

关于提示中帮助文本的一件棘手的事情是:必须将这些文本放在靠近鼠标悬停的元素旁。为此,需要使用 `tooltipNode` 获取组件的位置。通过在 DOM 节点上设置 `offsetTop` 和 `offsetLeft` 来调整提示文本的位置。这些都是 HTML 标准的 DOM 节点属性(<https://developer.mozilla.org/en-US/docs/Web/API/Node>),不是 React 的内容:

```
top: tooltipNode.offsetTop,
left: tooltipNode.offsetLeft
}))
},
```

代码清单 10.4 中是 `toggle()` 函数的完整代码(`ch10/tooltip/script.jsx`)。

#### 代码清单 10.4 `toggle()` 函数

```
toggle() {
  const tooltipNode = ReactDOM.findDOMNode(this)
  this.setState({
    opacity: !this.state.opacity,
    top: tooltipNode.offsetTop,
    left: tooltipNode.offsetLeft
  })
}
```

下面是使用 ES 的解构赋值写法:

```
toggle() {  
  const {offsetTop: top, offsetLeft: left} = ReactDOM.findDOMNode(this)  
  
  this.setState({  
    opacity: !this.state.opacity,  
    top,  
    left  
  })  
}
```

阅读上述代码, 可以看到状态和属性变了。现在需要重新渲染视图, 对吗? 不, 因为 React 将会帮助更新视图。setState() 会自动调用一次, 进行重新渲染。这是否会改变 DOM, 取决于在下一节要实现的 render() 中用到的状态。

### 10.2.2 render() 函数

render() 函数既包含用于帮助文本的 CSS 样式对象, 也包含 Twitter Bootstrap 样式。首先, 需要定义样式对象。需要根据 this.state.opacity 的值来设置 CSS 样式 opacity 和 z-index。需要用 z-index 来使帮助文本可以浮动在任何其他元素上, 所以将值设置得适当高一些, 当文本需要显示时为 1000, 当不需要显示时为 -1000:

```
zIndex: (this.state.opacity) ? 1000 : -1000,
```

对于 z-index, 需要使用 zIndex (注意使用驼峰命名规范)。图 10.2 显示了当鼠标悬停时 (opacity 是 true), 样式是如何生效的。

**提示:** 记得在 React 中使用驼峰命名规范代替连字符语法。将 CSS 属性 z-index 改为 React 样式属性 zIndex, 将 background-color 改为 backgroundColor, 将 font-family 改为 fontFamily, 等等。如果使用了有效的 JavaScript 名称, React 就可以从虚拟 DOM 更快速地更新真实 DOM。

状态的透明度属性 this.state.opacity 为布尔值 true 或 false, 但是 CSS 的透明度为二进制 0 或 1。如果状态的透明度是 false, CSS 的透明度就是 0; 如果状态的透明度是 true, CSS 的透明度就是 1。这需要使用一个二进制操作符(+)进行转换:

```
opacity: +this.state.opacity
```

就提示工具的位置而言, 希望将帮助文本放置在鼠标悬停的文本附近。方法是设置 top (窗口的顶部边缘到元素的距离) 为 20px, 并设置 left (窗口的左侧边缘到元素的距离) 为 -30px。这些值可以根据合适的逻辑随意调整:

```
render() {  
  const style = {  
    zIndex: (this.state.opacity) ? 1000 : -1000,  
    opacity: +this.state.opacity,  
    top: (this.state.top || 0) + 20,
```

```

left: (this.state.left || 0) - 30
}

```

鼠标悬停时，改变样式以显示增加的文本

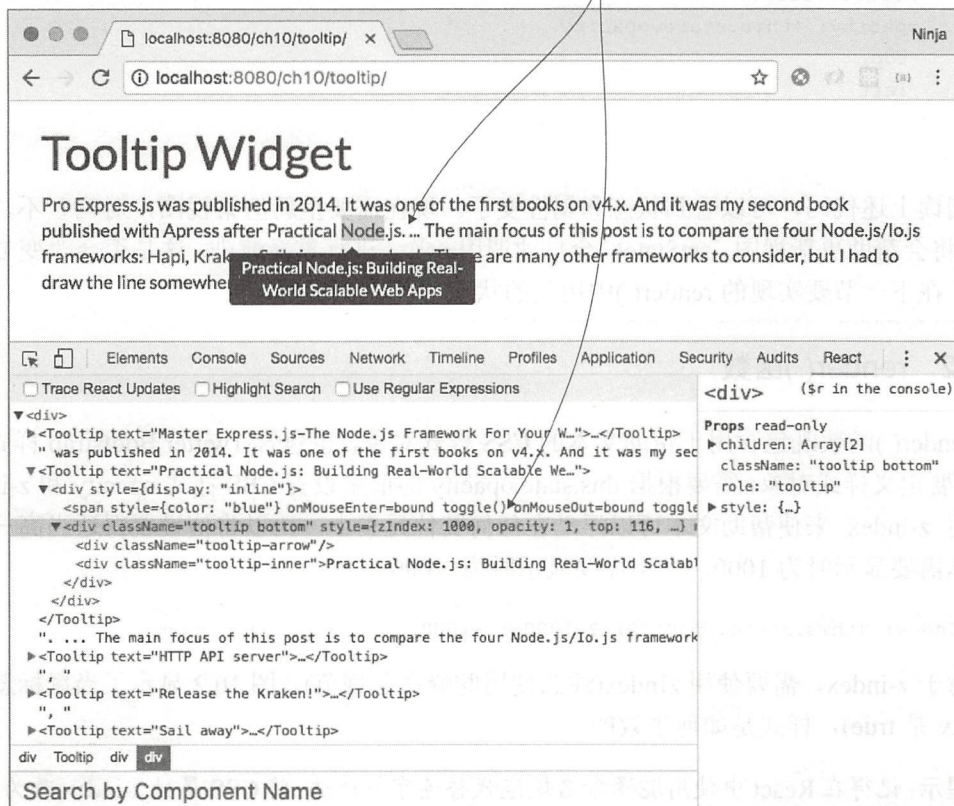


图 10.2 通过设置 opacity 的值为 1、zIndex 的值为 1000，使鼠标悬停时显示帮助文本

接下来是 `return`。组件将会一并渲染需要悬停的文本和帮助文本。将 Twitter Bootstrap 类和自有的样式对象一起使用以隐藏帮助文本，并在稍后显示。

在用户悬停时可以看到提示文本是蓝色的，因此可以将它们在视觉上与其他文本区分开来。当鼠标进入和离开 `span` 标签时，有两个鼠标事件：

```

return (
  <div style={{display: 'inline'}}>
    <span style={{color: 'blue'}}
      onMouseEnter={this.toggle}
      onMouseOut={this.toggle}
      {this.props.children}
    </span>

```

无论输出什么到HTML中，都将会在稍后传递给Tooltip组件

接下来是帮助文本的代码。它们像是静态的，除了 `{style}`。React 将会更改状态，并且会触发 UI 的更改：



使用 tooltip-arrow 类来实现一个指示箭头

```

<div className="tooltip bottom" style={style} role="tooltip">
  <div className="tooltip-arrow"></div>
  <div className="tooltip-inner">
    {this.props.text}
  </div>
</div>

```

将样式对象应用到 style 属性

从 text 属性(this.props.text) 输出提示的文本

代码清单 10.5 展示了 Tooltip 组件完整的 render() 方法。

代码清单 10.5 Tooltip 组件的完整的 render() 方法

```

render() {
  const style = {
    zIndex: (this.state.opacity) ? 1000 : -1000,
    opacity: +this.state.opacity,
    top: (this.state.top || 0) + 20,
    left: (this.state.left || 0) - 30
  }
  return (
    <div style={{display: 'inline'}}>
      <span style={{color: 'blue'}}
        onMouseEnter={this.toggle}
        onMouseOut={this.toggle}>
        {this.props.children}
      </span>
      <div className="tooltip bottom"
        style={style}
        role="tooltip">
        <div className="tooltip-arrow"></div>
        <div className="tooltip-inner">
          {this.props.text}
        </div>
      </div>
    </div>
  )
}

```

将 opacity、zIndex 和基于位置的合适位移应用到 DOM 节点

在鼠标移入时触发帮助文本的显示

将传入的任何文本作为 Tooltip 组件的内容输出

使用 Twitter Bootstrap 类输出帮助文本

至此，你已经完成了 Tooltip 组件！

## 10.3 运行 Tooltip 组件

使用 npm 编译这些 JSX 代码，试用或在项目中使用此组件：

```
$ npm run build
```

这个 Tooltip 组件很酷，感谢 Twitter Bootstrap 样式。也许它不像其他模块那样通用，但它是从头开始构建的。这正是我想要的！在 Twitter Bootstrap 类和 React 的帮助下，几乎

可以立即创建工作良好的工具提示(参见图 10.3)。它甚至是响应式的: 由于采用动态定位, 它可以适应各种尺寸的屏幕!

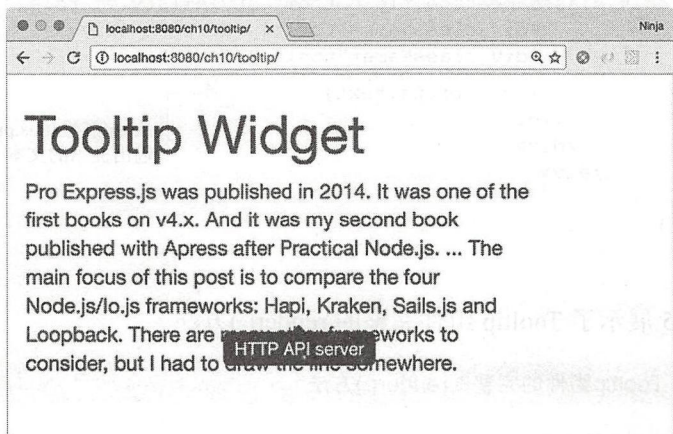


图 10.3 当用户把鼠标悬停在蓝色的文本上时, 被黑色容器包围的文本和指示箭头出现了, 提供了额外的信息

## 10.4 测验

为了有更多收获, 不妨尝试下面的操作:

- 创建响应鼠标单击的另一工具提示方案, 当单击高亮显示的文本时显示工具提示, 当再次单击文本时隐藏工具提示。
- 通过添加属性增强 Tooltip, 确定是悬停行为还是单击行为。
- 通过添加属性增强 Tooltip, 定位帮助文本到高亮文本的上方, 而不是默认显示在高亮文本的下方(提示: 更改 TB 类, 并更改 top 和 left 属性)。

提交代码到 ch010 目录下的一个新文件夹中, 作为本书的 GitHub 仓库(<https://github.com/azat-co/react-quickly>)的合并请求。

## 10.5 小结

- React 样式属性采用驼峰命名规范, 与 CSS 样式属性不同。
- `this.props.children` 包含组件的内容。
- 不需要手动重新渲染, 因为 React 会在调用 `setState()` 后自动重新渲染。

# 第 11 章

## 项目：Timer 组件

### 本章内容：

- 理解项目结构与脚手架
- 构建应用的架构

研究表明，冥想对于健康(平静)和生产力(专注)<sup>1</sup>非常重要。谁不希望变得更健康，更有生产力，特别是用最少的货币投资？

专家建议，开始时只要 5 分钟的冥想，然后延长到 10 分钟，最后在几周后延长到 15 分钟。目标是每天进行 30 至 60 分钟的冥想，但是有些人每天只需要 10 分钟就可以观察到改进。可以证明：在 3 年内每天冥想 10 分钟，会变得更加专注，并且在其他方面也有帮助。

但是，怎么才能知道达到了每天的冥想目标？这需要使用计时器！因此在本章中，将使用 React 和 HTML5 技术创建并测试 Web 定时器(请参阅图 11.1)。为了便于测试，这个定时器只能运行 5 秒、10 秒和 15 秒。

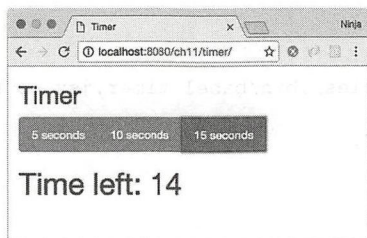


图 11.1 运行中的计时器示例，剩余 14 秒，一秒钟之前单击了 15seconds 按钮

1 参见“Research on Meditation”，维基百科：[https://en.wikipedia.org/wiki/Research\\_on\\_meditation](https://en.wikipedia.org/wiki/Research_on_meditation)；“Meditation:In Depth”，National Institutes of Health：<http://mng.bz/01om>；“Harvard Neuroscientist: Meditation Not Only Reduces Stress, Here’s How It Changes Your Brain”，《华盛顿邮报》2015 年 5 月 26 日：<http://mng.bz/1ljZ>；以及“Benefits of Meditation”，Yoga Journal：<http://mng.bz/7Hp7>



思路是：由三个控制器控制倒数计时器(从  $n$  到 0)。考虑一个典型的厨房计时器，它会数秒，而不是数分钟。单击一个按钮，计时器开始。再次单击，或单击另一个按钮，计时器重新开始。

注意：为跟进这个项目，需要下载未压缩的 React 版本，并安装 `node.js` 和 `npm` 来编译 JSX。在这个例子中，将使用 Bootswatch(<https://bootswatch.com/flatly>)中名为 Flatly 的主题。这个主题依赖 Twitter Bootstrap。附录 A 介绍了如何安装它们。

## 11.1 项目结构和脚手架

Timer 组件的项目结构与 Tooltip 和 Menu 组件的不同，如下所示：

```
/timer
  /node_modules
  bootstrap.css
  flute_c_long_01.wav
  index.html
  package.json
  react-dom.js
  react.js
  timer.js
  timer.jsx
```

← Babel开发依赖，用于将JSX转为JS

← 标识时间结束的音频文件

← 主JSX脚本

与前面一样，有一个 `node_modules` 文件夹用来存放开发依赖，例如用来将 JSX 转换为 JS 的 Babel。项目结构是扁平化的，样式和脚本都在同一个文件夹内。这样做是为了保持简单。在真实的应用中，会将样式和脚本放在不同的文件夹里。

`package.json` 中的关键部分是 `npm` 构建脚本、Babel 配置、依赖和其他元数据，参见代码清单 11.1。

代码清单11.1 Timer项目的package.json文件

```
{
  "name": "timer",
  "version": "1.0.0",
  "description": "",
  "main": "script.js",
  "scripts": {
    "build": "./node_modules/.bin/babel timer.jsx -o timer.js -w"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": ["react"]
  },
  "devDependencies": {
    "babel-cli": "6.9.0",
    "babel-preset-react": "6.5.0"
  }
}
```

← 创建一个npm脚本，将JSX转换为JS

无论是通过复制和粘贴，还是通过直接输入来创建 `package.json`，在这之后，一定要运

行 `npm i` 或 `npm install`。

这个项目的 HTML 是非常基础的，参见代码清单 11.2(ch11/timer/index.html)，里面包含 `react.js` 和 `react-dom.js` 文件。为了保持简单，这些都和 HTML 在相同的文件夹内。

代码清单11.2 Timer项目的Index.html文件

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Timer</title>
    <script src="react.js" type="text/javascript"></script>
    <script src="react-dom.js" type="text/javascript"></script>
    <link href="bootstrap.css" rel="stylesheet" type="text/css"/>
  </head>
  <body class="container-fluid">
    <div id="timer-app"/>
  </body>
  <script src="timer.js" type="text/javascript"></script>
</html>
```

这个文件只包含库并指向从 `timer.jsx` 创建的 `timer.js`。为此，需要 Babel CLI(参见第 3 章)。

## 11.2 应用架构

`timer.jsx` 文件由三个组件组成：

- **TimerWrapper**：主要组件，承担大部分工作并渲染其他部分。
- **Timer**：显示剩余秒数的组件。
- **Button**：渲染三个按钮并触发(重置)Timer 组件。

图 11.2 显示了它们将在页面中如何显示。可以看到 **Timer** 和 **Button** 组件，**TimerWrapper** 组件包含全部三个按钮和 **Timer** 组件。**TimerWrapper** 组件是一个容器(智能组件)，另外两个都是展示组件(木偶组件)。

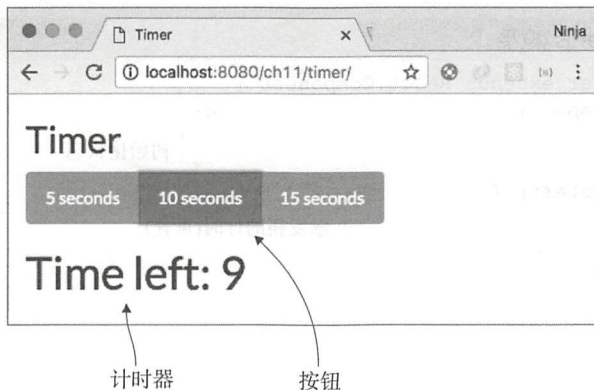


图 11.2 Timer 和 Button 组件

我们将应用分成三部分，因为在软件工程中，随着每次发布更新，事情往往会很快发生变化。通过分离表现(Button 和 Timer 组件)和逻辑(TimerWrapper 组件)，可以使应用更具适应性，并且还可以在其他应用中重复使用按钮等元素。本质上，保持表现和业务逻辑分离是使用 React 的最佳实践。

需要 TimerWrapper 组件来使 Timer 和 Buttons 组件保持通信。这三个组件和用户之间的交互如图 11.3 所示：

- ① TimerWrapper 组件通过将自身的状态作为属性传递给 Timer 和 Buttons 组件来渲染它们。
- ② 用户使用按钮来交互，这会在按钮上触发一个事件。
- ③ 按钮上的事件调用 TimerWrapper 组件中的函数，并将时间(秒数)作为参数。
- ④ TimerWrapper 组件设置循环并更新 Timer 组件。
- ⑤ 持续更新，直到只剩 0 秒。

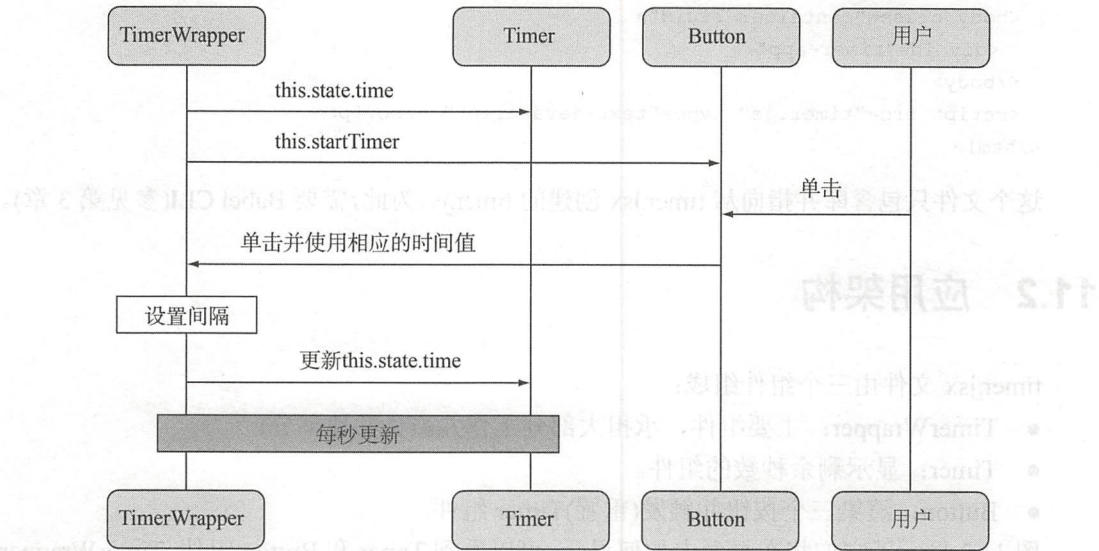


图 11.3 计时器应用的执行，在顶部开始

为了简单起见，可以保持三个组件都在 timer.jsx 文件中，参见代码清单 11.3。

代码清单11.3 timer.jsx概要

```
class TimerWrapper extends React.Component {
  constructor(props) {
    // ...
  }
  startTimer(timeLeft) {
    // ...
  }
  render() {
    // ...
  }
}

class Timer extends React.Component {
  render() {
    // ...
  }
}
```

初始化状态

触发新的计时(重置)



```
// ...  
}
```

```
class Button extends React.Component {  
  startTimer(event) {  
    // ...  
  }  
  render() {  
    // ...  
  }  
}
```

← 由用户的单触发新的  
计时(重置), 调用  
TimerWrapper组件的  
startTimer()函数

```
ReactDOM.render(  
  <TimerWrapper/>,  
  document.getElementById('timer-app')  
)
```

← 渲染TimerWrapper组件

让我们从 timer.jsx 文件的下方开始并渲染主要组件(TimerWrapper)到 id 为 timer-app 的 <div>元素中:

```
ReactDOM.render(  
  <TimerWrapper/>,  
  document.getElementById('timer-app')  
)
```

ReactDOM.render() 将会是此文件中的最后一个调用。它使用了 TimerWrapper 组件, 所以接下来定义这个组件。

## 11.3 TimerWrapper 组件

TimerWrapper 组件是实现所有功能的地方! 下面是对该组件的高级概览:

```
class TimerWrapper extends React.Component {  
  constructor(props) {  
    // ...  
  }  
  startTimer(timeLeft) {  
    // ...  
  }  
  render() {  
    // ...  
  }  
}
```

首先, 需要能够存储剩余时间(使用 timeLeft 属性)并重置计时器(使用 timer 属性)。因此, 使用两个状态属性: timerLeft 和 timer。

当应用首次加载完成时, 计时器不需要运行。所以在 TimerWrapper 组件的 constructor() 中, 需要设置时间值(timeLeft 属性)为 null。这将在 Timer 组件中派上用场, 因为需要区分首次加载(timeLeft 值为 null)和计时结束(timeLeft 值为 0)。

同样还要设置状态属性 timer 的值为 null。timer 属性保存了对倒计时函数 setInterval()

的引用。但是由于现在没有运行计时器，因此值为 `null`。

最后，绑定 `startTimer()` 函数，因为要将它作为(按钮的)事件处理程序：

```
class TimerWrapper extends React.Component {
  constructor(props) {
    super(props)
    this.state = {timeLeft: null, timer: null}
    this.startTimer = this.startTimer.bind(this)
  }
  ...
}
```

接下来是 `startTimer` 事件处理程序。用户每次单击按钮时都会调用它。当用户单击某个按钮时，如果计时器已经在运行，那么需要清除之前的循环并开始一个新的循环。因为很明确，不需要多个计时器同时工作。因此，需要在 `startTimer()` 方法中做的第一件事是：通过清除 `setInterval()` 的结果来停止上一倒计时。当前计数器的 `setInterval` 对象存储在 `this.state.timer` 变量中。

`clearInterval()` 方法可以清除 `setInterval()` 的结果。`clearInterval()` (<http://mng.bz/7104>) 和 `setInterval()` (<http://mng.bz/P2d6>) 都是浏览器 API 方法。即它们可以通过 `window` 对象使用，而不需要添加额外的库或前缀等(将 `window.clearInterval()` 写在浏览器中，代码能够正常执行，但是在 Node.js 中会中止)。在按钮的事件处理程序的第一行调用 `clearInterval()`：

```
class TimerWrapper extends React.Component {
  constructor(props) {
    // ...
  }
  startTimer(timeLeft) {
    clearInterval(this.state.timer)
    // ...
  }
}
```

在清除前一个计时器之后，可以通过 `setInterval()` 设置一个新的计时器。传递给 `setInterval()` 的代码每秒都会被调用。下面这段代码使用一个箭头函数来绑定 `this` 到当前上下文，绑定后，可以在 `setInterval()` 的(回调)函数中使用 `TimerWrapper` 组件的状态、属性和方法：

```
class TimerWrapper extends React.Component {
  constructor(props) {
    // ...
  }
  startTimer(timeLeft) {
    clearInterval(this.state.timer)
    let timer = setInterval(() => {
      // ...
    }, 1000)
    // ...
  }
  render() {
    // ...
  }
}
```

接下来实现此函数。timeLeft 变量表示计时器上剩余的时间，用来保存对当前值减 1 的结果并检查是否到了 0。如果到了，就通过调用 clearInterval()，并使用存储在 timer 变量中的计时器对象(通过 setInterval() 创建)的引用来移除计时器。此引用在 setInterval() 结束时保存，使未来的函数可以调用(每秒循环一次)。这是 JavaScript 作用域的工作方式。所以，没必要从状态中获取计时器引用的值(虽然可以)。

接下来，在每次循环间隔中保存 timeLeft 的值。并且到最后，当单击按钮时，保存 timeLeft 和 timer 对象：

```
//...
startTimer(timeLeft) {
  clearInterval(this.state.timer)
  let timer = setInterval(() => {
    var timeLeft = this.state.timeLeft - 1
    if (timeLeft == 0) clearInterval(timer)
    this.setState({timeLeft: timeLeft})
  }, 1000)
  return this.setState({timeLeft: timeLeft, timer: timer})
}
//...
```

使用 setState() 设置状态为新的值，这个过程是异步的。setInterval() 间隔长度是 1000 毫秒或 1 秒。需要设置状态的 timeLeft 和 timer 为新的值，因为应用更新需要这些值，并且此处不能使用简单的变量或属性。

setInterval() 被 JavaScript 的事件循环异步调度执行。setState() 的返回值会在 setInterval() 的第一次回调前触发。这一点可以在代码中简单地通过使用控制台日志输出来验证。例如，下列代码将会打印 1，然后打印 2；而不是先打印 2，然后打印 1：

```
...
startTimer(timeLeft) {
  clearInterval(this.state.timer)
  let timer = setInterval(() => {
    console.log('2: Inside of setInterval')
    var timeLeft = this.state.timeLeft - 1
    if (timeLeft == 0) clearInterval(timer)
    this.setState({timeLeft: timeLeft})
  }, 1000)
  console.log('1: After setInterval')
  return this.setState({timeLeft: timeLeft, timer: timer})
}
...
```

最后，TimerWrapper 组件必须有 render() 函数。它返回 <h2>、三个按钮和 Timer 组件。row-fluid 和 btn-group 都是 Twitter Bootstrap 类：它们使按钮更漂亮，虽然这对 React 来说并不重要：

```
render() {
  return (
    <div className="row-fluid">
      <h2>Timer</h2>
```



```

<div className="btn-group" role="group" >
  <Button time="5" startTimer={this.startTimer}/>
  <Button time="10" startTimer={this.startTimer}/>
  <Button time="15" startTimer={this.startTimer}/>
</div>

```

这段代码展示了如何通过为 `time` 属性提供不同的值来重用 `Button` 组件。这些 `time` 属性值允许按钮在各自的标签上显示不同的时间并设置不同的计时器。所有按钮的 `startTimer` 属性值都相同。值为来自 `TimerWrapper` 组件的 `this.startTimer`，如前所述，它可以启动/重置计时器。

接下来，需要显示文本“Time left: ...”，它由 `Timer` 组件渲染。为此，将状态 `time` 作为属性传给 `Timer` 组件。为了保持 React 最佳实践，`Timer` 组件是无状态的。当属性被状态(`TimerWrapper` 组件)的更改更新时，React 自动更新页面(`Timer` 组件)上的文本。稍后再实现 `Timer` 组件。现在，像下面这样使用它：

```
<Timer time={this.state.timeLeft}/>
```

另外，当计时器开始时，`<audio>` 标签(一种 HTML5 标签，指向文件)会提醒你：

```

<audio id="end-of-time" src="flute_c_long_01.wav" preload="auto">
  </audio>
</div>
)
}
}

```

为了参考和更好地加以理解(有时很高兴看到整个组件)，查看一下定时器应用，`TimerWrapper` 组件的完整代码如代码清单 11.4 所示(ch11/timer/timer.jsx)。

代码清单 11.4 `TimerWrapper` 组件

```

class TimerWrapper extends React.Component {
  constructor(props) {
    super(props)
    this.state = {timeLeft: null, timer: null}
    this.startTimer = this.startTimer.bind(this)
  }
  startTimer(timeLeft) {
    clearInterval(this.state.timer)
    let timer = setInterval(() => {
      console.log('2: Inside of setInterval')
      var timeLeft = this.state.timeLeft - 1
      if (timeLeft == 0) clearInterval(timer)
      this.setState({timeLeft: timeLeft})
    }, 1000)
    console.log('1: After setInterval')
    return this.setState({timeLeft: timeLeft, timer: timer})
  }
  render() {
    return (
      <div className="row-fluid">
        <h2>Timer</h2>
        <div className="btn-group" role="group" >
          <Button time="5" startTimer={this.startTimer}/>

```

重置时清除计时器，以防万一有其他计时器正在运行

每秒更新递减后剩余的时间

使用不同的时间调用 `startTimer` 来渲染按钮

```

    <Button time="10" startTimer={this.startTimer}/>
    <Button time="15" startTimer={this.startTimer}/>
  </div>
  <Timer timeLeft={this.state.timeLeft}/>
  <audio id="end-of-time" src="flute_c_long_01.wav"
    preload="auto"></audio>
</div>
)
}
}

```

渲染文本 "Time left:..."，并在到达 0 时播放声音

当时间为 0 时，HTML5 的 <audio> 标签播放提示

TimerWrapper 组件有很多逻辑。其他组件都是无状态组件，并且基本上没有逻辑。但是还有另外两个组件需要实现。不要忘了 TimerWrapper 组件中的 <audio> 标签，当剩余时间到达 0 时将播放声音。接下来，让我们实现 Timer 组件。

## 11.4 Timer 组件

Timer 组件的目标是显示剩余时间，并且当时间到了播放声音。Timer 是一个无状态组件。实现这个类，并检查 timerLeft 属性是否等于 0：

```

class Timer extends React.Component {
  render() {
    if (this.props.timeLeft == 0) {
      // ...
    }
    // ...
  }
}

```

要播放声音(flute\_c\_long\_01.wav 文件)，需要为项目使用特殊的 HTML5 元素 <audio>；需要在 TimerWrapper 组件中使用它，并且将 src 指向 WAV 文件，设置 id 为 end-to-time。所有需要做的事情是获取 DOM 节点(Javascript 方法 getElementById() 会很好用)并且执行 play() (同样是 HTML5 的 Javascript 方法)。这些再一次说明，如果足够勇敢，React 可以与 HTML5、jQuery 3<sup>2</sup> 甚至 Angular 4 融合得很好：

```

class Timer extends React.Component {
  render() {
    if (this.props.timeLeft == 0) {
      document.getElementById('end-of-time').play()
    }
    // ...
  }
}

```

如前所述，开始并不希望计时器显示 0，因为计时器尚未运行。所以，在 TimerWrapper 组件(参见代码清单 11.4)中，初始化时设置 timeLeft 的值为 null。如果 timerLeft 为 null 或 0，那么 Timer 组件将渲染一个空白的 <div>。意味着应用不会显示 0：

```

if (this.props.timeLeft == null || this.props.timeLeft == 0)
  return <div/>

```

2 例如集成浏览器事件和 jQuery，详见第 6 章



另外, 当 `timeLeft` 大于 0 时, 用 `<h1>` 元素显示剩余时间。换言之, 当计时器运行时, 需要显示剩余时间:

```
return <h1>Time left: {this.props.timeLeft}</h1>
```

为便于引用, 下面的代码清单 11.5 给出了 `Timer` 组件的全部代码(ch11/timer/timer.jsx)。

代码清单 11.5 `Timer` 组件, 显示剩余时间

```
class Timer extends React.Component {
  render() {
    if (this.props.timeLeft == 0) {
      document.getElementById('end-of-time').play()
    }
    if (this.props.timeLeft == null || this.props.timeLeft == 0)
      return <div/>
    return <h1>Time left: {this.props.timeLeft}</h1>
  }
}
```

← 当时间结束时播放声音

← 显示文本“Time left:...”

初始化时什么都不显示

由于 `Timer` 组件需要显示秒数, 因此需要先启动计时器, 这会在单击按钮时发生。

## 11.5 Button 组件

为了遵守 DRY(Don't Repeat Yourself, 不要重复自己)原则<sup>3</sup>, 创建一个 `Button` 组件, 并使用三次来显示三个不同的按钮。`Button` 是另一类无状态(且很简单)组件, 所以它应该符合 `React` 的要求, 但是 `Button` 组件不像 `Timer` 组件那样简单, 因为 `Button` 组件包含事件处理程序。

按钮必须有 `onClick` 事件处理程序来捕获用户单击按钮行为。这些单击会触发计时器进行倒计时。启动计时器的函数并非在 `Button` 组件中实现, 而是在 `TimerWrapper` 组件中实现并由父组件向下传递给 `Button` 组件, 使用的函数是 `TimerWrapper` 组件的 `this.props.startTimer`。但是如何将时间(5、10 或 15)传给 `TimeWrapper` 组件的 `startTimer`? 来看看位于 `TimerWrapper` 组件的下面这段代码, 它通过属性传递时间值:

```
<Button time="5" startTimer={this.startTimer}/>
<Button time="10" startTimer={this.startTimer}/>
<Button time="15" startTimer={this.startTimer}/>
```

思路是通过这个组件渲染三个按钮(代码重用)。为了知道用户选择了哪个时间, 需要将 `this.props.time` 的值作为参数传给 `this.props.startTimer`。

如果写了如下代码, 它们将不会执行:

```
// 不会执行, 必须是定义
<button type="button" className='btn-default btn'
```

3 DRY 原则如下: “每一项知识必须在系统中具有单一、明确、权威的表述”; 见维基百科 <http://mng.bz/1K5k> 以及 Andrew Hunt 所著的《程序员修炼之道: 从小工到专家》(<http://amzn.to/2ojjXoY>)





恭喜——至此已经正式完成了编码。现在，让这些代码运行起来，这样就可以在工作<sup>4</sup>中或为兴趣爱好使用这个计时器了。

## 11.6 运行 Timer 组件

使用下面的 Babel 6.9.5 命令编译 JSX 为 JavaScript，假设已经有了 Babel CLI 并且预设已经安装(提示：package.json!)：

```
$ ./node_modules/.bin/babel timer.jsx -o timer.js -w
```

如果从本章开头的 package.json 复制 npm 构建脚本，那么可以运行 `npm run build`。

如果所有的事情都操作正确，就请欣赏这个漂亮的计时器应用吧，如图 11.4 所示！关闭音乐，以便可以听到时间结束时的提示。

保证应用工作正常：应该看到显示剩余时间的数字每秒都在更改。当单击按钮时，会重新开始倒计时；就是这样，每次单击按钮，计时器被终止并重新开始。

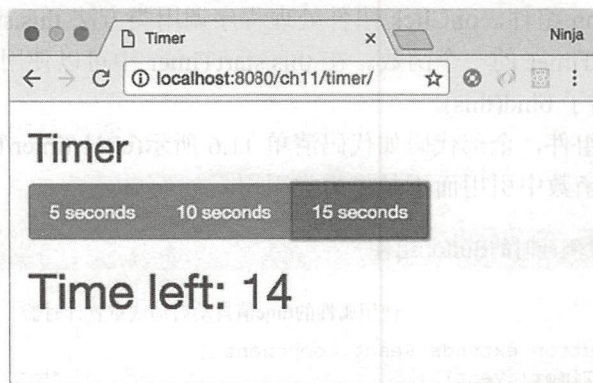


图 11.4 单击“15 seconds”按钮来运行计时器，现在显示剩余 14 秒

## 11.7 测验

为了有更多收获，不妨尝试下面的操作：

- 将 Timer 转换为由箭头函数实现的无状态组件。
- 实现一个暂停/恢复按钮来暂停/恢复计时器。
- 实现一个取消按钮来停止倒计时并隐藏剩余时间。
- 实现一个重置按钮来重置剩余时间为初始值(5、10 或 15 秒)。
- 修改这个项目的最终版本，使用 5、10 和 15 分钟，而不是秒。
- 解耦 TimerWrapper 组件中的 `<audio>` 标签与 Timer 组件中的 `play()`。
- 重构这个项目到 4 个文件中：timer.jsx、timer-label.jsx、timer-button.jsx 和 timer-sound.jsx，尽量松散。

4 尝试番茄工作法(<https://cirillocompany.de/pages/pomodoro-technique>)来提高效率

- 实现一个滑块，在每个时间间隔改变(第 6 章讨论了滑块集成)。

提交代码到 ch11 目录下的一个新文件夹中,作为本书的 GitHub 仓库(<https://github.com/azat-co/react-quickly>)的合并请求。

## 11.8 小结

- 保持组件简单并尽量具有具象性。
- 将函数作为属性值而不仅仅是数据传递。

两个组件可以通过父组件交换数据。



## 第 II 部分

# React 架构

欢迎来到本书第 II 部分。你现在已经了解了 React 最重要的概念、特性和模式，已经准备好开始自己的 React 旅程。本书第 I 部分使你做好了构建简单 UI 元素的准备；原则是，如果正在构建 Web UI，使用 React 核心就足够了。但是为了构建完整的前端应用，React 开发者依赖 React 社区编写的开源模块。这些模块中的大部分都托管在 GitHub 和 npm 上，所以它们都很容易获取——可以下载它们并使用。

本书第 II 部分包含的这些章涵盖最受欢迎、最常用的成熟的库，它们与 React 核心一起构成了 React 技术栈(正如一些开发者笑称此集合为：React 全家桶)。在开始前，第 12 至第 17 章将介绍如何使用 Webpack 组织资源管道，使用 React Router 管理 URL 路由，使用 Redux 和 GraphQL 管理数据流，使用 Jest 进行测试，以及使用 Express 和 Node 做同构 React 应用。然后，类似本书第 I 部分，第 18 至第 20 章介绍真实的项目。

这可能看起来内容很多，但是基于我多年阅读和写作书籍的经验，婴儿学步般的步骤和教科书般的例子并不能为读者提供有效价值，也不能说明现实生活中的事情。所以，在本书的第 II 部分，你将学习和使用 React 技术栈。有趣的、复杂的项目在等着你。完成这些项目之后，你将对数据流有所了解，能熟练地设置被称为 Webpack 的怪物，并在聚会时能够像无所不知的家伙那样侃侃而谈。

请继续阅读。

# 第 12 章

## Webpack 构建工具

本章内容：

- 在项目中添加 Webpack
- 模块化代码
- 运行 Webpack 并测试构建
- 执行热模块替换

在我们深入 React 技术栈(又称 React 全家桶)之前，来看看现在绝大部分 Web 开发的基础：构建工具(或称 bundler)。可以在接下来的几章中使用这个工具，将代码文件打包为运行应用所需的最小数量的文件，并准备好它们以便于部署。这个将要使用的构建工具就是 Webpack(<https://webpack.js.org>)。

如果之前从未遇见过构建工具，或者使用了其他的构建工具，如 Grunt、Gulp 或 Bower，那么本章就是为你准备的。你将学习如何建立和配置 Webpack，以及用它运行项目。

本章还涵盖了热模块替换(Hot Module Replacement, HMR)，这是 Webpack 的一个特性，可以在实时服务器上热替换已更新的模块。首先，我们来看看 Webpack 可以为你做些什么。

注意：代码生成器，如 create-react-app(<https://github.com/facebookincubator/create-react-app>)，可创建样板/脚手架代码并帮助快速启动项目。create-react-app 同样使用 Webpack 和 Babel 以及其他模块。但是本书只讲基本知识，所以不需要使用代码生成器；相反，你将自己设置项目来保证理解每一部分。如果感兴趣，可以自学如何使用代码生成器——只需要几个命令。

### 12.1 Webpack 的作用

是否思考过为什么 Web 开发中的所有人都在讨论 Webpack？Webpack 的核心重点是优化

编写的 JavaScript，使其包含在尽量少的文件中以便于客户端请求。这减小了热门网站的服务器压力，也减小了客户端的页面加载时间。当然，并不像上面所讲的那么简单。JavaScript 经常被编写成易于重用的模块，但是它们经常依赖可能还会再依赖另一些模块的其他模块，等等；Webpack 还会跟踪什么需要加载，以便令人头痛的依赖关系问题可以迅速解决。

例如有一个实用程序模块 `myUtil`，要在很多 React 组件中使用，比如 `accounts.jsx`、`transactions.jsx` 等。如果没有像 Webpack 这样的工具，事实上就必须手动跟踪，每次使用这些组件中的某个，都需要包含 `myUtil` 作为依赖。此外，可能会加载 `myUtil` 第二次或第三次，有些是没必要的，因为依赖 `myUtil` 的另一个组件已经加载它了。当然，这是一个简单的例子，真正的项目有几十个甚至数百个在其他依赖关系中使用的依赖关系。Webpack 可以解决这些问题。

Webpack 知道如何处理所有三种类型的 JavaScript 模块——CommonJS([www.commonjs.org](http://www.commonjs.org))、AMD(<https://github.com/amdjs/amdjs-api/wiki/AMD>)和 ES6(<http://mng.bz/VjyO>)——所以不需要担心混合使用了不同的模块类型。Webpack 将会分析项目中的所有 JavaScript 依赖关系并执行以下操作：

- 确保所有依赖都以正确的顺序加载。
- 确保所有的依赖都仅加载一次。
- 确保 JavaScript 被打包到尽量少的文件中(称为静态资源)。

Webpack 也支持代码分割和资源哈希，这可以识别仅在特定情况下才需要的代码块。这些模块被拆分出来按需加载，而不是和其他所有都打包在一起。必须选择使用这些功能，并进一步优化 JavaScript 及其部署。

注意：代码分割和资源哈希超出了本书的范围。访问 Webpack 网站以获取更多信息：<https://webpack.github.io/docs/code-splitting.html>。

Webpack 并不仅仅关于 JavaScript，通过使用加载器，它还支持其他静态文件的预处理。例如，可以在所有打包之前做如下处理：

- 将 JSX、Jade 或 CoffeeScript 文件预编译为普通的 JavaScript。
- 将 ES6 以上的代码预编译到 ES5 中，因为浏览器现在还不支持 ES6。
- 将 Sass 和 Compass 文件预编译为 CSS。
- 将精灵图优化为单独的 PNG 或 JPG 文件，或优化为内联数据资源。

许多加载器可以用于多种文件类型。另外，Webpack 首页上已经对将会修改 Webpack 行为的插件进行了分类。如果找不到想要的插件，有关于如何编写自定义插件的文档可供参考。

在本书的剩余部分里，将使用 Webpack 完成下列事情：

- 从 npm 模块管理包依赖。这样就不需要从网络手动下载文件并使用 `<script>` 标签将它们包含在 HTML 中。
- 将 JSX 编译为常规的 JavaScript，同时提供源代码映射以便于调试。
- 管理样式。
- 实现热模块替换。
- 建立一台开发 Web 服务器。



综上所述,可以使用 `webpack.config.js` 配置 Webpack 加载、预编译和文件的打包顺序。但是首先,我们来看看如何安装并在项目中使用 Webpack。

## 12.2 添加 Webpack 到项目中

为了说明如何开始使用 Webpack,让我们稍微修改图 12.1 所示的第 7 章中的项目。它具有电子邮件和评论输入字段、两个样式表和一个 Content 组件。

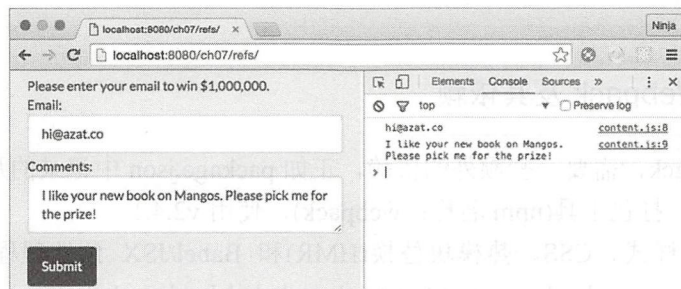
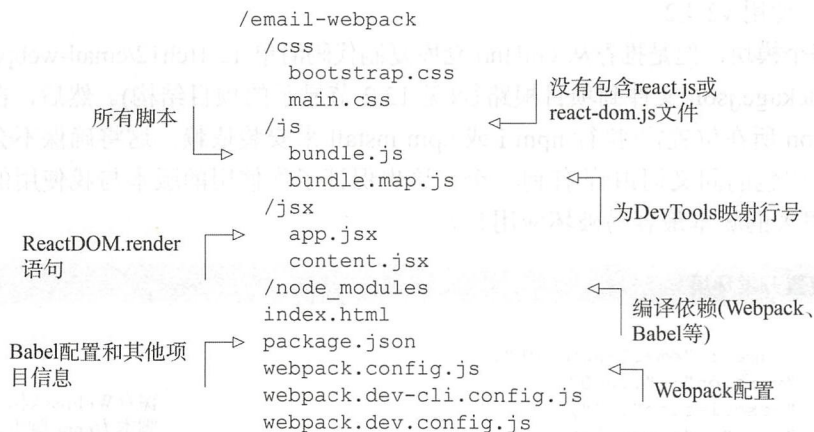
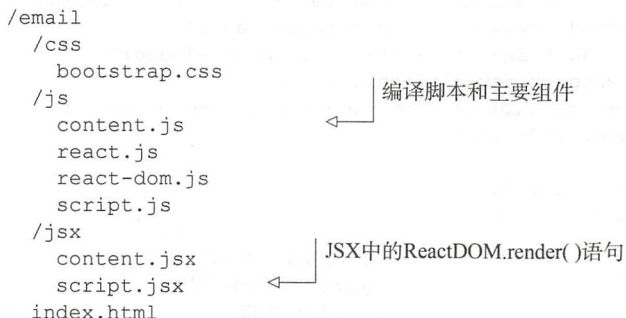


图 12.1 使用 Webpack 前的原始电子邮件项目

下面是新的项目结构。我已经指出与第 7 章中项目的不同之处:



与第 7 章中没有使用 Webpack 的项目结构作对比:



注意：是否有 Node.js 和 npm？现在是安装它们的最佳时间——需要它们才可以继续。附录 A 介绍了如何安装它们。

本节将引导完成以下步骤：

- ① 安装 Webpack。
- ② 安装依赖并将它们保存到 package.json 中。
- ③ 配置 Webpack 的 webpack.config.js。
- ④ 配置开发服务器和热模块替换。

我们开始吧。

### 12.2.1 安装 Webpack 及其依赖

要使用 Webpack，需要一些额外的依赖，正如 package.json 中记录的那样：

- Webpack：打包工具(npm 名称：webpack)，使用 v2.4.1。
- Loaders：样式、CSS、热模块替换(HMR)和 Babel/JSX 预处理程序(npm 名称：style-loader、css-loader、react-hot-loader、babel-loader、babel-core 和 babel-preset-react)，使用的版本在 package.json 中指定。
- webpack-dev-server：一台 Express 开发服务器，可以使用 HMR(npm 名称：webpack-dev-server)，使用 v2.4.2。

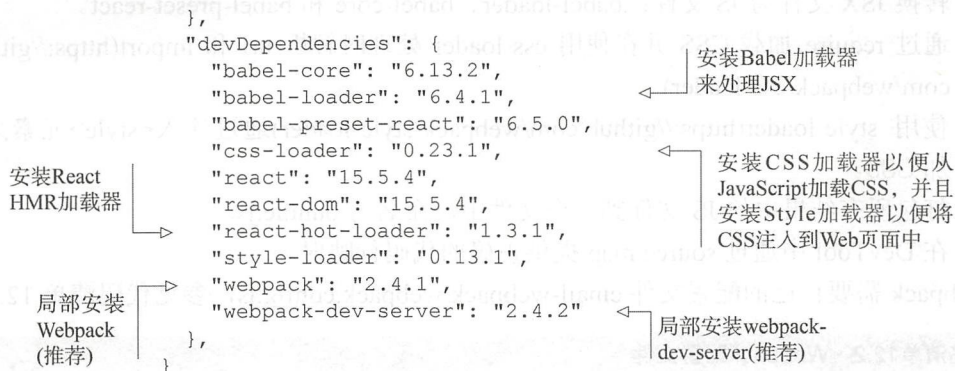
可以手动安装每个模块，但是推荐从 GitHub 仓库复制代码清单 12.1(ch12/email-webpack/package.json)中的 package.json 文件到项目根路径(见 12.2 节显示的项目结构)。然后，在项目根路径(package.json 所在位置)下执行 npm i 或 npm install 来安装依赖。这将确保不会遗漏 10 个模块(Node 中包的同义词)中的任何一个。这也保证了你使用的版本与我使用的足够接近。使用差异很大的版本最容易破坏应用！

代码清单12.1 设置开发环境

```
{
  "name": "email-webpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "build": "./node_modules/.bin/webpack -w"
    "wds-cli": "./node_modules/.bin/webpack-dev-server --inline --hot
      --module-bind 'css=style-loader!css-loader'
      --module-bind 'jsx=react-hot-loader!babel-loader'
      --config webpack.dev-cli.config.js",
    "wds": "./node_modules/.bin/webpack-dev-server --config
      webpack.dev.config.js"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"
    ]
  }
}
```

保存Webpack构建脚本为npm脚本，作为快捷方式

告诉Babel需要使用什么预设(此处对于JSX是React, ES6+可选)



`package.json` 中的 `babel` 属性与本书第 1 章中的相似, 所以不再赘述。提示: 需要使用这个属性来配置 Babel 以便转换 JSX 为 JS。如果支持那些不支持 ES6 的浏览器, 可以添加 `es2015` 预设到 `presets` 中:

```
"babel": {
  "presets": [
    "react",
    "es2015"
  ]
},
```

也可以添加 `babel-preset-es2015` 到 `devDependencies` 中:

```
"devDependencies": {
  "babel-preset-es2015": "6.18.0",
  ...
}
```

在新的依赖之外, 还有新的 `npm` 脚本。`package.json` 中的 `scripts` 命令是可选的, 但是强烈建议使用, 因为在使用 `React` 和 `Node` 时使用 `npm` 脚本来运行和构建是最佳实践。当然, 也可以手动运行所有的构建而不是使用 `npm` 脚本, 但是为什么要输入额外的字符呢?

可以使用 `npm run build` 或直接使用 `./node_modules/.bin/webpack-w` 运行 Webpack。-w 标记的意思是观察——持续监听任何源代码的改变, 如有更改, Webpack 重新构建模块。当然, 必须通过 `npm i` 安装所有必需的模块。

`webpack-w` 命令默认寻找 `webpack.config.js`。还不能用这个配置来运行 Webpack。接下来我们创建它。

注意: `package.json` 中的 `wds` 和 `wds-cli` `npm` 脚本将在 12.5 节中解释。

## 12.2.2 配置 Webpack

Webpack 需要知道要处理什么(源代码)以及如何去做处理(使用加载器)。这就是在项目结构的根路径下需要 `webpack.config.js` 的原因。概括来说, 在此项目中, 使用 Webpack 来做以下事情:



- 转换 JSX 文件为 JS 文件: babel-loader、babel-core 和 babel-preset-react。
- 通过 require 加载 CSS 并在使用 css-loader 处理时解析 url 和 import(<https://github.com/webpack/css-loader>)。
- 使用 style-loader(<https://github.com/webpack/style-loader>)通过注入<style>元素来添加 CSS。
- 打包所有结果中的 JS 文件到一个文件中, 命名为 bundle.js。
- 在 DevTool 中通过 source map 提供正确的代码行映射。

Webpack 需要自己的配置文件 `email-webpack/webpack.config.js`, 参见代码清单 12.2。

代码清单12.2 Webpack配置文件

```
module.exports = {  
  entry: './jsx/app.jsx',  
  output: {  
    path: __dirname + '/js/',  
    filename: 'bundle.js'  
  },  
  devtool: '#sourcemap',  
  module: {  
    loaders: [  
      { test: /\.css$/, loader: 'style-loader!css-loader' },  
      {  
        test: /\.jsx?$/,  
        exclude: /(node_modules)/,  
        loaders: ['babel-loader']  
      }  
    ]  
  }  
}
```

为打包完的文件定义路径

定义开始打包的文件(典型的是加载其他文件的主文件)

为打包完的文件定义文件名以便在index.html中使用

指定从编译过的源码行到JSX源码行的需要的、合适的映射。这在调试和在DevTool中显示时非常有用

指定加载器来从JavaScript导入, 并注入CSS到Web页面中

指定加载器来执行JSX转换(和ES6+, 如果需要的话)

`devtool` 属性在开发中非常有用, 因为它提供了源码映射, 可以告诉你源代码而非编译后代码中的行号。现在已经准备好为此项目运行 Webpack, 并且将来还可以引导任何基于 Webpack 的项目。

### 配置文件

如果想要, 可以有不止一个配置文件。这些文件可以在开发、生产、测试和其他构建中派上用场。在该例的项目结构中, 创建了这些文件:

```
webpack.dev-cli.config.js
```

```
webpack.dev.config.js
```

只要你和团队成员能够理解每个文件的含义, 如何命名就没关系。名称可以通过 `--config` 传给 Webpack。将在 12.4 节中学习关于这些配置文件的更多信息。

Webpack 有很多功能, 我们只是介绍了基础知识, 但是它们已足以编译 JSX、提供源码映射、注入和导入 CSS 以及打包 JavaScript。当需要更多的 Webpack 功能时, 可以查阅文档或 Juho Vepsäläinen 编写的图书 *SurviveJS*(<https://survivejs.com>)。

现在已经准备好在 JSX 中使用 Webpack 的一些功能了。

## 12.3 模块化代码

如前所述,在第 7 章,电子邮件应用使用了全局对象和<script>。这对于本书或小型应用来说很好。但是在大型应用中,使用全局变量不被接受,因为可能遇到命名冲突或管理多个包含重复内容的<script>标签的麻烦。可以通过 CommonJS 语法让 Webpack 管理所有依赖。Webpack 将会只包含需要的依赖并将它们打包到 bundle.js 文件(基于 webpack.config.js 中的配置)中。

通过模块化来组织代码,不仅是适用于 React,也是适用于一般软件工程的最佳实践。可以使用 Browserify、SystemJS 或另一个 bundler/module 加载器,并依然使用 CommonJS/Node.js 语法(require 和 module.exports)。因此,本节中的代码可以移植到其他系统,只需要将其重构为全局模式而非原来的模式。

截至撰写本书时,import(http://mng.bz/VjyO)只受浏览器 Edge 支持,不受 Node.js 支持。使用 import 语法的 ES6 模块在 Webpack 的配置中将需要做更多的工作。它不是 CommonJS 的 require/module.exports 语法的确切替代者,因为这些命令工作不同。因此,下面的代码清单 12.3 使用 require() 和 module.exports 代替全局对象和 HTML<script> 标签重构了 app.jsx (ch12/email-webpack/app.jsx)。由于使用了 style-loader,因此能够请求 CSS 文件;并且因为使用了 Babel 加载器,所以也能够请求 JSX 文件。

代码清单 12.3 重构 app.jsx

```
require('../css/main.css')

const React = require('react')
const ReactDOM = require('react-dom')
const Content = require('./content.jsx')

ReactDOM.render(
  <Content />,
  document.getElementById('content')
)
```

导入 CSS, 感谢 style-Loader 和 css-loader, CSS 将被导入并注入到 Web 页面中

导入 React 以便使用 < 语法: React.createElement()

导入 Content 组件

作为对比, ch07/email/jsx/script.jsx 看起来如下:

```
ReactDOM.render(
  <Content />,
  document.getElementById('content')
)
```

旧文件更小一些,但这是为数不多的情况之一。它依赖全局的 Content、ReactDOM 和 React 对象,对此,正如我刚刚解释的,是糟糕的实践。

在 content.jsx 中,可以用相同的方式使用 require()。constructor()、submit() 和 render() 的代码没有变化:

```
const React = require('react')
const ReactDOM = require('react-dom')
```

← 导入 React  
← 导入 ReactDOM

```
class Content extends React.Component {
  constructor(props) {
    // ...
  }
  submit(event) {
    // ...
  }
  render() {
    // ...
  }
}
```

```
module.exports = Content ← 导出 Content
```

index.html 文件需要指向 Webpack 创建的 bundle: js/bundle.js 文件。它的名字是在 webpack.config.js 中指定的, 现在需要添加它。它将会在运行 npm run build 后被创建。下面是新的 index.html 中的代码:

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/bootstrap.css" type="text/css" rel="stylesheet"/>
  </head>
  <body>
    <div id="content" class="container"></div>
    <script src="js/bundle.js"></script>
  </body>
</html>
```

注意, 对 main.css 的引用也被从 index.html 中移除了。Webpack 将注入一个带有 main.css 引用的 <style> 元素到 index.html 中, 因为在 app.jsx 中有 require('main.css')。也可以使用 require() 引用 bootstrap.css。

以上是重构项目的最后一步。

## 12.4 运行 Webpack 并测试构建

现在是验证的时刻。运行 \$npm run build 并与下面的结果比较输出:

```
> email-webpack@1.0.0 build
  /Users/azat/Documents/Code/react-quickly/ch12/email-webpack
> webpack -w
Hash: 2ffe09fff88a4467788a
Version: webpack 1.12.9
Time: 2545ms
```

Asset	Size	Chunks	Chunk Names
bundle.js	752 kB	0 [emitted]	main



```
bundle.js.map      879 kB      0 [emitted] main
+ 177 hidden modules
```

如果没有错误，就能看到在 js 文件夹中新创建的 bundle.js 和 bundle.js.map 文件。现在打开你最喜欢的 Web 服务器(可能是 node-static 或 http-server)，并访问此 Web 应用。你会看到它正在控制台中记录电子邮件和注释。

由此可见，将 Webpack 整合到项目中是很简单的，并且会产生很好的效果。

### 177 个隐藏的模块或被遮盖的 Webpack 包

在 ch12/email-webpack/js/bundle.js 中有 177 个模块！可以打开此文件并搜索 webpack\_require(1)、webpack\_require(2)等，直到 webpack\_require(176)，它们是 Content 组件。接下来这些编译的来自 app.jsx 的代码导入了 Content 组件(bundle.js 的第 49 至第 53 行)：

```
const React = __webpack_require__(5);
const ReactDOM = __webpack_require__(38);
const Content = __webpack_require__(176);

ReactDOM.render(React.createElement(Content, null),
  document.getElementById('content'));
```

至少，已经做了一些准备，可以在本书的其余部分使用 Webpack 了。但是强烈建议安装另一个东西：可以显著加速开发的热模块替换(HMR)。在我们继续 React 开发之前，让我们来看一下这个重要的 Webpack 功能。

### ESLint 和 Flow

我想提一下另外两个有用的开发工具。显然，它们是可选的，但是它们非常重要。

ESLint(<http://eslint.org>, npm 名称为 eslint)可以使用预定义的规则或规则集，并确保代码(JS 或 JSX)遵守相同的标准。例如，多少个空格是一个缩进——四个还是两个？或者，如果不小心在代码中加了一个分号，会发生什么呢(分号在 JavaScript 中是可选的，我不喜欢使用它们)？ESLint 甚至会给出关于未使用变量的警告。它可以防止错误偷偷溜进代码(当然不是全部)！

查看文章“Getting Started with ESLint”(<http://eslint.org/docs/user-guide/gettingstarted>)。你还需要 eslint-plugin-react(<https://github.com/yannickcr/eslint-plugin-react>)。确保已经将 React 规则添加到 .eslintrc.json 中(全部代码在 ch12/email-webpack-eslint-flow 文件夹中)：

```
"rules": {
  "react/jsx-uses-react": "error",
  "react/jsx-uses-vars": "error",
}
```

以下是在 ch12/email-webpack-lint-flow/jsx/content.jsx 上运行 ESLint React 的一些警告示例：

```
/Users/azat/Documents/Code/react-quickly/ch12/
└─ email-webpack-lint-flow/jsx/content.jsx
   9:10 error 'event' is defined but never used no-unused-vars
  12:5   error Unexpected console statement no-console
  12:17 error Do not use findDOMNode react/no-find-dom-node
  13:5   error Unexpected console statement no-console
```



13:17 error Do not use findDOMNode

react/no-find-dom-node

接下来, Flow(<https://flowtype.org>, npm 名称为 flow-bin)是一个静态类型检查工具, 通常可以添加特殊的注释(`//@flow`)到类型和脚本中。是的! 在 JavaScript 中使用类型! 如果你是一名喜欢 Java、Python 和 C 等强类型语言的软件工程师, 那么这值得庆祝。一旦添加了注释, 就可以运行 Flow 来检查是否有任何问题。再次, 这个工具可以防止一些讨厌的错误:

```
// @flow
var bookName: string = 13
console.log(bookName) // number. This type is incompatible with string
```

Flow 有大量的文档可供参考, 参见文章“Getting started with Flow”(<https://flowtype.org/docs/getting-started.html>)和“Flow for React”(<https://flowtype.org/docs/react.html>)。

可以配置 Atom 或任何其他的现代代码编辑器以使用 ESLint 和 Flow 来动态捕获问题, 如图 12.2 所示。

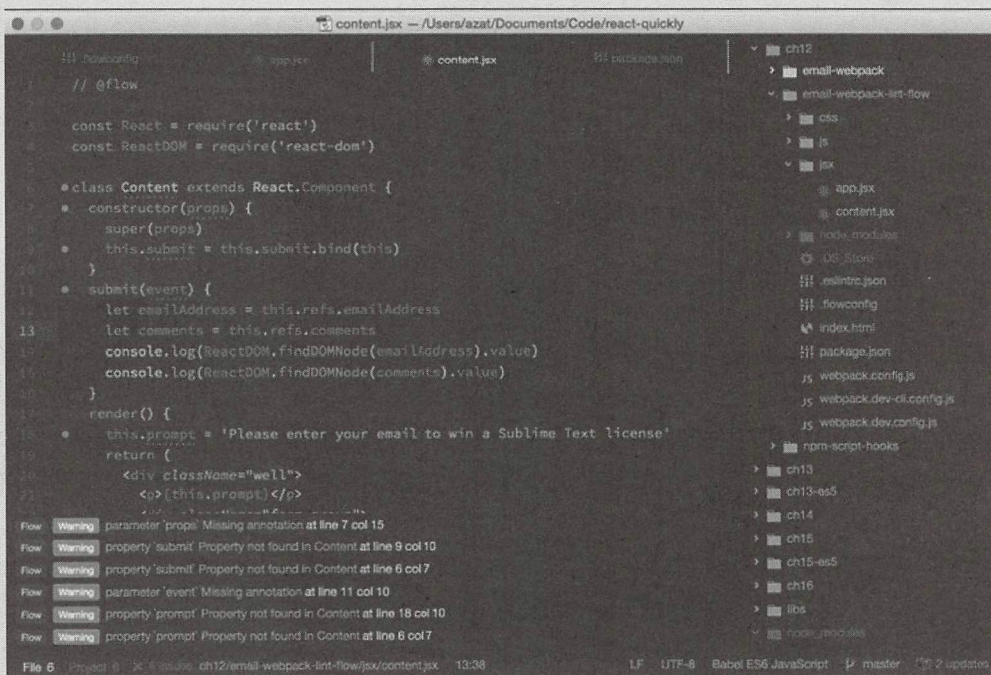


图 12.2 Atom 代码编辑器支持 Flow, 在开发中, 它会在底部的面板上显示问题并在代码所在行进行标记

可以在 `ch12/email-webpack-eslint-flow` 文件夹中找到带 ESLint v3.8.1 和 Flow v0.33.0 的电子邮件项目代码。

## 12.5 热模块替换

热模块替换(HMR)是 Webpack 和 React 最酷的功能之一。它可以让你编写代码, 并通过将更改更新至浏览器来更快地测试它, 同时保持应用的状态。





点、public 路径和 HMR 插件，并将 dev-server 标记设置为 true。代码清单 12.4 显示了最终文件(ch12/email-webpack/webpack.dev.config.js)。

代码清单12.4 webpack-dev-server和HMR配置

```
const webpack = require('webpack')
module.exports = {
  entry: [
    'webpack-dev-server/client/?http://localhost:8080',
    './jsx/app.jsx'
  ],
  output: {
    publicPath: 'js/',
    path: __dirname + '/js/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',
  module: {
    loaders: [
      { test: /\.css$/, loader: 'style-loader!css-loader' },
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loaders: ['react-hot-loader', 'babel-loader']
      }
    ]
  },
  devServer: {
    hot: true
  },
  plugins: [new webpack.HotModuleReplacementPlugin()]
}
```

导入webpack模块

包含WDS

包含主应用

为WDS设置路径，使bundle.js可用(这不会写到磁盘)

包含react-hot-loader以使HMR自动应用到所有JSX文件

设置WDS为HMR模式

包含HMR插件

需要通过提供--config 选项来告诉 WDS 使用新的配置文件：

```
./node_modules/.bin/webpack-dev-server --config webpack.dev.config.js
```

如果还没有保存，保存到 package.json 中是为了方便，记得添加 react-hot-loader 到 dependencies 节点。此模块使 HMR 作用于所有的 JSX 文件(后者又被转换为 JS 文件)。

我倾向于使用 react-hot-loader 为所有的文件开启 HMR。但是如果仅希望为特定的模块而不是所有的模块开启 HMR，不要使用 react-hot-loader；作为替代，可选择手动将 module.hot.accept() 语句添加到为 HMR 选择的 JSX/JS 模块。此魔法般的 module.hot 模块来自 Webpack。建议检查 module.hot 是否可用：

```
if(module.hot) {
  module.hot.accept()
}
```

这么多配置！但是还有另一种使用和配置 Webpack 的方法：可以使用命令行选项并在命令中包含一些配置。

如果喜欢使用命令行，悉听尊便。配置文件会更小，但命令会更长。例如，webpack.dev-cli.config 文件的配置较少：

```

module.exports = {
  entry: './jsx/app.jsx',
  output: {
    publicPath: 'js/',
    path: __dirname + '/js/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loaders: []
      }
    ]
  }
}

```

但使用了更多的 CLI 选项:

```

./node_modules/.bin/webpack-dev-server --inline --hot
➡ --module-bind 'css=style-loader!css-loader'
➡ --module-bind 'jsx=react-hot-loader!babel-loader'
➡ --config webpack.dev-cli.config.js

```

此处发生了几件事情。首先, --inline 和 --hot 在入口上开启了 WDS 和 HMR 模式。然后, 使用 --module-bind(通过如下语法)将加载器传递过去:

```
fileExtension=loader1!loader2!...
```

确保 react-hot 在 babel 之前; 否则, 会发生错误。

使用 CLI 还是完整的配置文件, 看个人选择。我发现更简单的构建方式是使用 CLI。为了避免以后发现错误输入复杂命令时的痛苦, 应该把命令保存为 package.json 中的 npm 脚本。并且批处理/shell 脚本/Make 脚本不再炫酷了。使用 npm 脚本, 像所有的炫酷潮人一样!

### npm 脚本

npm 脚本提供了一定的优势, 而且它们通常用在 Node 和 React 项目中。它们已经成为事实上的标准, 当第一次了解一个项目时, 通常会找到它们。当我开始一个新的项目或者库时, npm 脚本是我在看过 readme.md 之后要看的第一个地方——有时会代替 readme.md, readme.md 可能会过时。

npm 脚本提供了一种灵活的方式来保存必要的脚本, 用于测试、构建、散播数据, 以及在开发或其他环境中运行。换言之, 任何通过 CLI 执行的与应用相关的工作, 但不是应用本身, 都可以保存到 npm 脚本中。它们也起到文档的作用, 向其他人展示如何进行构建和测试工作。可以从 npm 脚本中调用其他 npm 脚本, 从而进一步简化项目。下列示例包含不同版本的构建:



```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "build": "./node_modules/.bin/babel -w",
  "build:method": "npm run build -- method/jsx/script.jsx -o
    ➤ method/js/script.js",
  "build:hello-js-world-jsx": "npm run build --
    ➤ hello-js-world-jsx/jsx/script.jsx -o
    ➤ hello-js-world-jsx/js/script.js",
  "build:hello-world-jsx": "npm run build --
    ➤ hello-world-jsx/jsx/script.jsx -o
    ➤ hello-world-jsx/js/script.js",
  "build:hello-world-class-jsx": "npm run build --
    ➤ hello-world-class-jsx/jsx/script.jsx -o
    ➤ hello-world-class-jsx/js/script.js"
},
```

npm 脚本也支持 pre 和 post 钩子(hook)，这可以使它们更加通用。一般来说，钩子是一种模式，其中的一些代码在另一事件发生时被触发。例如，可以创建一个连同两个 pre 和 post 钩子(prelearn-react 和 postlearn-react)的 learn-react 任务。如你所想，pre 钩子会在 learn-react 前执行，post 钩子会在 learn-react 后执行。例如，下面这些 bash 脚本：

```
"scripts": {
  "prelearn-react": "echo \"Purchasing React Quickly\"",
  "learn-react": "echo \"Reading React Quickly\" ",
  "postlearn-react": "echo \"Creating my own React app\""
},
```

基于 pre/post 顺序打印如下输出：

```
...
Purchasing React Quickly
...
Reading React Quickly
...
Creating my own React app
```

通过使用 pre 和 post 钩子，npm 可以轻松替换由 Webpack、Gulp 或 Grunt 执行的一些构建步骤。

参考 <https://docs.npmjs.com/misc/scripts> 上的文档和 Keith Cirkel 的文章 “How to Use npm as a Build Tool” ([www.keithcirkel.co.uk/how-to-use-npm-as-a-build-tool](http://www.keithcirkel.co.uk/how-to-use-npm-as-a-build-tool)) 来获取更多的 npm 提示，包括参数。npm 脚本缺少的任何功能都可以作为 Node 脚本从头开始实现。好处是项目的插件依赖较少。

## 12.5.2 热模块替换实践

继续使用 `npm run wds` 或 `npm run wds-cli` 启动 WDS。然后，转到 `http://localhost:8080` 并打开开发者工具的控制台。你将看到来自 HMR 和 WDS 的消息，如下所示：

```
[HMR] Waiting for update signal from WDS...
```



```
[WDS] Hot Module Replacement enabled
```

在电子邮件或评论字段中输入一些文本，然后修改 `content.jsx`。可以在 `render()` 中修改一些东西——例如，修改表单文本，从 `Email` 变为 `Your Email`：

```
Your Email: <input ref="emailAddress" className="form-control" type="text"
  placeholder="hi@azat.co"/>
```

你将会看到如下日志：

```
[WDS] App updated. Recompiling...
...
[HMR] App is up to date
```

然后，更改将显示在网页上，如图 12.4 所示，页面上还显示了之前输入的文本。太棒了——不需要再浪费时间输入测试数据或深入嵌套的 UI！可以花更多的时间做重要的事情，而不是输入和单击前端应用。使用 HMR 时开发速度更快！

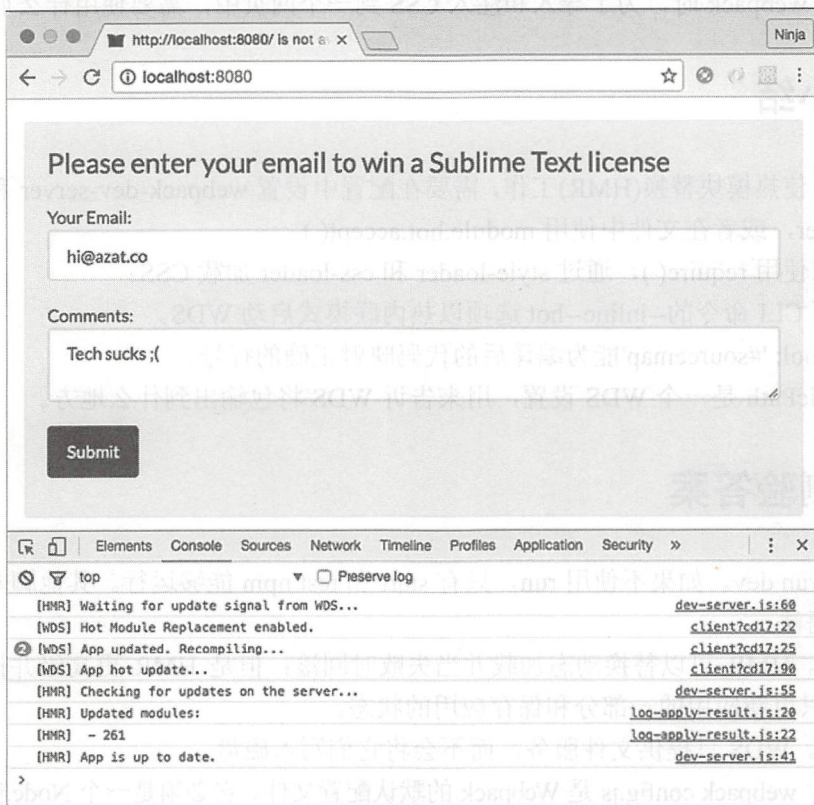


图 12.4 HMR 在不清除字段中数据的情况下更新视图，从“Email”更新为“Your Email”，如日志中所示

**注意：**HMR 不是完全可靠的。在某些情况下，它不会更新甚至可能失败。此时 WDS 将重新加载页面(实时重新加载)。此行为由 `webpack/hot/dev-server` 控制；另一个选择是使用 `webpack/hot/only-dev-server` 手动重新加载。

Webpack 是能与 React 一起使用的一个很好的工具，可以简化和增强打包。WDS 和

HMR 不仅有助于部署时优化代码、图像、样式和其他资源，而且对开发也很有帮助。感谢 WDS 和 HMR。

## 12.6 测验

1. 运行 npm 脚本 dev 的命令是什么("dev": "./node\_modules/.bin/webpack-dev-server --config webpack.dev.config.js")? npm dev、npm run dev、NODE\_ENV=dev npm run 还是 npm run development?
2. HMR 只是用于 React 动态加载的一个术语，这么理解正确还是错误?
3. WDS 会将编译后的文件写入磁盘，和 webpack 命令一样，这么理解正确还是错误?
4. webpack.config.js 必须是一个有效的 JSON 文件，就像 package.json 一样，这么理解正确还是错误?
5. 使用 Webpack 时，为了导入和注入 CSS 到一个网页中，需要使用什么加载器?

## 12.7 小结

- 为了使热模块替换(HMR)工作，需要在配置中设置 webpack-dev-server 和 react-hot-loader，或者在文件中使用 module.hot.accept()。
- 可以使用 require()，通过 style-loader 和 css-loader 加载 CSS。
- 使用 CLI 命令的--inline--hot 选项以热内联模式启动 WDS。
- devtool: '#sourcemap'能为编译后的代码映射正确的行号。
- publicPath 是一个 WDS 设置，用来告诉 WDS 将包输出到什么地方。

## 12.8 测验答案

1. npm run dev。如果不使用 run，只有 start 和 test npm 能够运行。其他脚本遵循 npm run NAME 语法。
2. 错误。HMR 可以替换动态加载并当失败时回滚；但是 HMR 更高级并提供更多的好处，例如只更新应用的一部分和保存应用的状态。
3. 错误。WDS 只提供文件服务，而不会将它们写入磁盘。
4. 错误。webpack.config.js 是 Webpack 的默认配置文件。它必须是一个 Node.js/JavaScript 文件，其 CommonJS/Node.js 模块导出的文本对象用作配置(该对象可以有双引号，类似于 JSON)。
5. style 加载器导入，CSS 加载器注入。

# 第 13 章

## React 路由

本章内容:

- 从零开始实现路由
- 使用 React Router
- 使用 Backbone 的路由

以前, 在很多单页面应用中, URL(如果有的话)很少会随着应用的使用而改变。没有请求服务器的理由, 感谢浏览器渲染! 页面上只有部分内容发生了变化。这种做法存在一些不良影响:

- 刷新浏览器会改变当前正在阅读的页面并回到初始状态。
- 单击“后退”按钮可能会回退到完全不同的站点, 因为浏览器的历史记录功能只记录所在站点的单个 URL, 并不反映在内容之间导航的 URL 变更。
- 无法与朋友分享网站上的精确页面。
- 搜索引擎不能为网站编制索引, 因为没有单独的 URL 来索引。

幸运的是, 今天有了浏览器 URL 路由。URL 路由允许将应用配置为接受不映射到物理文件的请求 URL。相反, 可以定义对用户而言在语义上有意义的 URL, 这可以帮助搜索引擎优化(Search-Engine Optimization, SEO), 并且可以反映应用程序的状态。例如, 显示产品信息页面的 URL 可能是:

<https://www.manning.com/books/react-quickly>

这会将后台整齐地映射到单个页面, 该页面显示了 id 为 react-quickly 的产品。当浏览各种产品时, URL 可能会改变, 浏览器和搜索引擎将能够按照期望与产品页面进行交互。如果想避免完整的页面重载, 可以像一些知名网站那样在 URL 中使用符号#:



```
https://mail.google.com/mail/u/0/#inbox
https://en.todoist.com/app?v=816#agenda%2Foverdue%2C%20today
https://calendar.google.com/calendar/render?tab=mc#main_7
```

URL 路由开发是用户体验良好且设计良好的 Web 应用的需求。没有特定的 URL，用户无法在不丢失应用的状态下，保存或共享链接，无论是单页面应用 (Single-Page Application, SPA) 还是由服务器呈现的传统 Web 应用。

在本章中，将构建一个简单的 React 网站，并了解在其中实现路由的几个不同选项。本章稍后会介绍 React Router 库；首先，让我们从头开始构建一些简单的路由。

## 13.1 从零开始实现路由

尽管有的库实现了 React 路由，但还是让我们从一个简单的路由开始，看看实现它是多么容易。这个项目也可以帮助理解其他路由如何工作。

该项目的最终目标是在浏览时，三个页面随 URL 一起更改。将使用 hash URL (#) 来保持简单；非 hash URL 需要特殊的服务器配置。下面这些是要创建的页面：

- Home——/(空 URL 路径)
- Accounts——/#accounts
- Profile——/#profile

图 13.1 展示了从 Home 页面到 Profile 页面的导航。

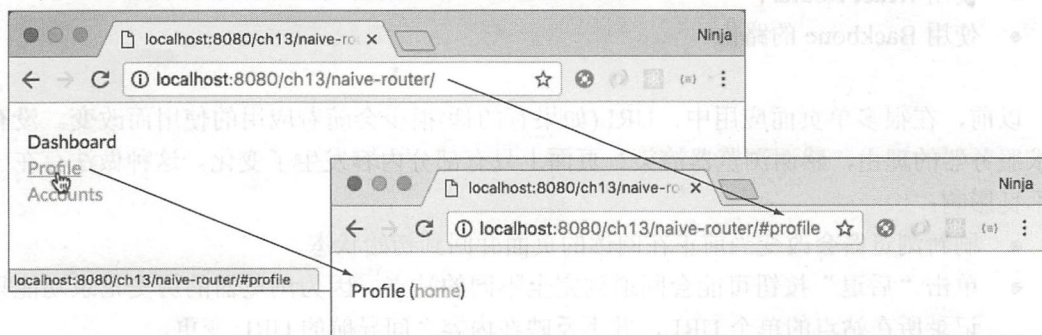


图 13.1 通过单击链接，从 Home 页面导航到 Profile 页面并修改 URL

为实现这个项目，该项目将展示和使用一个 URL 路由，需要创建一个路由组件 (router.jsx)、一个映射和一个 HTML 页面。路由组件将从 URL 获取信息并据此更新 Web 页面。该项目的实现过程可以分解为如下这些步骤：

① 写一个从浏览器输入的 URL 与所展示资源(React 元素和组件)间的映射。映射是应用特有的，每一个新项目都需要不同的映射。

② 从零开始写一个路由库。它将接收请求的 URL 并根据映射(步骤①)检查 URL。该路由库将会是 router.jsx 中的一个单独 Router 组件。该 Router 组件可以在不修改的情况下重用。

③ 写一个示例应用，它将使用步骤②中的 Router 组件和步骤①中的映射。

你将使用 JSX 标记语言来创建 React 元素。显然，Router 组件不必是 React 组件，也可以是常规的函数或类。但是使用 React 组件可以增强理解你在本书中学到的概念，例如生命周期事件，并利用 React 的 DOM 渲染和处理。另外，此实现将更接近 React Router 的实现，这将有助于更好地理解 React Router，我们稍后讨论。

### 13.1.1 建立项目

项目(可称为简单路由或原生路由)结构如下：

```
/naive-router
/css
  bootstrap.css
  main.css
/js
  bundle.js
/jsx
  app.jsx
  router.jsx
/node_modules
index.html
package.json
webpack.config.js
```

下面从安装依赖关系开始。将它们放在 package.json 中，可以复制依赖关系以及 babel 配置和脚本，并运行 npm install，如代码清单 13.1 所示(ch13/naive-router/package.json)。

代码清单13.1 搭建开发环境

```
{
  "name": "naive-router",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "./node_modules/.bin/webpack -w"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"
    ]
  },
  "devDependencies": {
    "babel-core": "6.18.2",
    "babel-loader": "6.2.4",
    "babel-preset-react": "6.5.0",
    "webpack": "2.4.1",
    "react": "15.5.4",
    "react-dom": "15.5.4"
  },
  "dependencies": {
  }
}
```

将Webpack构建脚本保存为npm脚本以提升便利性

告诉Babel使用什么预设  
(本例中使用适用于JSX的  
React, ES6+是可选的)

本地安装Webpack  
v2.4.1(推荐)

这不是全部。Webpack 需要自己的配置文件 `webpack.config.js`(如第 9 章所述), 如代码清单 13.2 所示。关键是配置源(输入)和目标(输出)。还需要提供加载器。

代码清单13.2 webpack.config.js

```

module.exports = {
  entry: './jsx/app.jsx',
  output: {
    path: __dirname + '/js/',
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
}

```

为打包完的文件定义文件名以便在index.html中使用

定义开始构建的文件(典型的是加载其他文件的主文件)

为打包完的文件定义路径

指定将执行JSX以及ES6+(如果需要的话)转换的加载器

### 13.1.2 在 app.jsx 中创建路由映射

首先, 将使用映射对象创建映射, 其中键是 URL 片段, 值是各个页面的内容。映射需要将一个值连接到另一个值。在这种情况下, 键(URL 片段)将映射到 JSX。可以为每个页面创建一个单独的文件, 但是现在把它们全部保存在 `app.jsx` 中, 如代码清单 13.3 所示。

代码清单13.3 路由映射(app.jsx)

```

const React = require('react')
const ReactDOM = require('react-dom')
const Router = require('./router.jsx')

const mapping = {
  '#profile': <div>Profile (<a href="#">home</a></div>,
  '#accounts': <div>Accounts (<a href="#">home</a></div>,
  '*': <div>Dashboard<br/>
    <a href="#profile">Profile</a>
    <br/>
    <a href="#accounts">Accounts</a>
  </div>
}

ReactDOM.render(
  <Router mapping = {mapping}>,
  document.getElementById('content')
)

```

使用CommonJS的require()来导入Webpack打包的模块

使用路由映射对象将路由映射到单独的页面

将映射传给路由



接下来，将在 `router.jsx` 中实现路由。

### 13.1.3 在 `router.jsx` 中创建 Router 组件

简而言之，Router 组件需要从 URL(`#profile`)获取信息，并使用提供给它的映射属性将其映射到 JSX。可以从浏览器 API 的 `window.location.hash` 访问 URL：

```
const React = require('react')
module.exports = class Router extends React.Component {
  constructor(props) {
    super(props)
    this.state = {hash: window.location.hash}
    this.updateHash = this.updateHash.bind(this)
  }
  render() {
    ...
  }
}
```

接下来，需要使用 `hashchange` 监听器监听任何 URL 变更。如果没有实现监听新的 URL，那么路由只能工作一次：在整个页面重新加载并且 Router 元素被创建时。添加和移除 `hashchange` 监听器的最佳位置是 `componentDidMount()` 和 `componentWillUnmount()` 生命周期事件监听函数：

```
updateHash(event) {
  this.setState({hash: window.location.hash})
}
componentDidMount() {
  window.addEventListener('hashchange', this.updateHash, false)
}
componentWillUnmount() {
  window.removeEventListener('hashchange', this.updateHash, false)
}
```

#### `componentDidMount()` 和 `componentWillUnmount()`

第 5 章讨论了生命周期事件。`componentDidMount()` 在元素被加载并出现在真实的 DOM 节点(可以说元素有真正的 DOM 节点)后被触发。因此，这是给其他 DOM 对象添加事件对象最安全的地方，也是进行 AJAX/XHR 调用(这里不使用)的最佳位置。

另一方面，`componentWillUnmount()` 是移除事件监听器的最佳位置。元素即将被卸载，并且需要删除在这个元素之外创建的任何东西(例如窗口上的一个事件监听器)。在移除的元素上留下很多事件监听器是十分糟糕的实践：会导致性能问题，如内存泄漏。

在 `render()` 中，使用 `if/else` 检查 `mapping` 属性中的键/属性是否与当前 URL 值(`this.state.hash`)匹配。如果匹配，可以再次访问 `mapping` 属性以获取单个页面(JSX)的内容。如果不匹配，为所有其他的 URL 回退到\*，包括空值(Home 页面)。完整的代码如代码清单 13.4 所示(`ch13/naive-router/jsx/router.jsx`)。

## 代码清单13.4 实现URL路由

```

const React = require('react')
module.exports = class Router extends React.Component {
  constructor(props) {
    super(props)
    this.state = {hash: window.location.hash}
    this.updateHash = this.updateHash.bind(this)
  }
  updateHash(event) {
    this.setState({hash: window.location.hash})
  }
  componentDidMount() {
    window.addEventListener('hashchange', this.updateHash, false)
  }
  componentWillUnmount() {
    window.removeEventListener('hashchange', this.updateHash, false)
  }
  render() {
    if (this.props.mapping[this.state.hash])
      return this.props.mapping[this.state.hash]
    else
      return this.props.mapping['*']
  }
}

```

分配初始的 URL hash值

提供新的 URL hash值

渲染与 URL hash 对应的内容

最后，在 index.html 中，包含在运行 `npm run build`(或运行 `./node_modules/.bin/webpack -w`)时 Webpack 将要生成的 CSS 文件和 bundle.js 文件：

```

<!DOCTYPE html>
<html>

  <head>
    <link href="css/bootstrap.css" type="text/css" rel="stylesheet"/>
    <link href="css/main.css" type="text/css" rel="stylesheet"/>
  </head>

  <body>
    <div id="content" class="container"></div>
    <script src="js/bundle.js"></script>
  </body>
</html>

```

运行打包器以获取 bundle.js，然后在浏览器中打开网页。单击链接可以改变 URL 和页面的内容，如之前的图 13.1 所示。

由此可知，用 React 构建自己的路由非常简单，可以使用生命周期方法来监听 hash 中的更改并呈现相应的内容。尽管这是可行的选择，但是如果需要嵌套路由、使用路由解析(提取 URL 参数)或使用不带#的 URL，情况会变得更加复杂。可以使用来自 Backbone 或另一个前端(MVC 类似框架)的路由，除此之外，还有专门为 React 设计的解决方案(提示：使用 JSX)。

## 13.2 React Router

React 在构建 UI 方面非常出色。如果不信服，请回头阅读前面的章！React 也可以用



来从头开始实现简单的 URL 路由，就像在 `router.jsx` 中看到的一样。

但是对于更复杂的 SPA 应用，则需要更多的功能。例如，传递 URL 参数是十分常用的特性，以此来表示单独的项而不是项的列表：例如 `/posts/57b0ed12fa81dea5362e5e98`，这里 `57b0ed12fa81dea5362e5e98` 是唯一的发布 id，可以使用正则表达式提取此 URL 参数；如果应用的复杂性增长了，迟早会发现需要改造已实现的用于前端的 URL 路由。

### 语义 URL

语义 URL 或好的 URL([https://en.wikipedia.org/wiki/Semantic\\_URL](https://en.wikipedia.org/wiki/Semantic_URL))旨在通过将内部实现与 UI 解耦来改进网站或 Web 应用的可用性和可访问性。非语义方法可能使用查询字符串或脚本文件名。另一方面，语义方式包含只使用路径，这在某种意义上能帮助用户解释结构和操作 URL。表 13-1 中是一些例子。

表 13-1 语义和非语义 URL 示例

非语义(可以)	语义(更好)
<code>http://webapplog.com/show?post=es6</code>	<code>http://webapplog.com/es6</code>
<code>https://www.manning.com/books/reactquickly?a_aid=a&amp;a_bid=5064a2d3</code>	<code>https://www.manning.com/books/reactquickly/a/5064a2d3</code>
<code>http://en.wikipedia.org/w/index.php?title=Semantic_URL</code>	<code>https://en.wikipedia.org/wiki/Semantic_URL</code>

主流框架(如 Ember、Backbone 和 Angular)都有内置的路由。说到路由和 React，React Router(`react-router`: <https://github.com/reactjs/react-router>)是已完成的、标准的解决方案。13.4 节介绍了 Backbone 实现，并说明了在这个很多人用于 SPA 的类 MVC 框架中，React 是如何良好工作的。现在，我们继续关注 React Router。

React Router 不是官方 React 核心库的一部分。它来自社区，但已经十分成熟且流行，三分之一的 React 项目在使用它<sup>1</sup>。对于我曾经交谈过的大多数 React 工程师来说，它是默认选择。

React Router 的语法使用 JSX，这是另一个优点，因为它允许创建比映射对象(正如你在前面项目中所见)更可读的分层定义。类似于原生路由的实现，React Router 有名为 Router 的 React 组件(React Router 启发了我的实现！)。可以遵循如下步骤：

① 创建一个映射，在该映射中，URL 将被转换为 React 组件(在网页上又被转换为标记)。在 React Router 中，和嵌套 Route，一样，这是通过传递 `path` 和 `component` 属性实现的。映射在 JSX 中通过声明和嵌套的 Route 组件完成。这部分在每个新项目中都必须实现。

② 使用 React Router 的 Router 和 Route 组件，它们魔法般根据 URL 的变化执行视图的更改。显然，不需要实现这部分，但是需要安装库。

③ 像普通 React 组件一样，通过使用 `ReactDOM.render()` 渲染 Router 到网页上。无须多言，这部分在每个新项目中都必须实现。

使用 JSX 为每个页面创建一个 Route，并将它们嵌套到另一个 Route 或 Router 中。将这个 Router 对象传递给 `ReactDOM.render()`，就像任何其他 React 组件一样：

<sup>1</sup> React.js Conf 2015, “React Router Increases Your Productivity”, 网址为 <https://youtube.com/watch?v=XZfvW1a8Xac>



```
ReactDOM.render((
  <Router ...>
    <Route ...>
      <Route .../>
      ...
    </Route>
    <Route .../>
  </Router>
), document.getElementById('content'))
```

每个 Route 至少有两个属性: path, 这是匹配到的触发该路由的 URL 模式; component, 它提取和渲染必要的组件。可以给 Route 设置更多的属性, 例如事件处理程序和数据。它们可以在 Route 组件的 props.route 中访问到。这就是给路由组件传递数据的方式。

举例说明, 考虑一个 SPA 示例, 路由到如下页面: About、Posts(类似于博客)、单独的 Post、Contact Us 和 Login。它们有不同的路径, 并且从不同的组件渲染:

- About—/about
- Posts—/posts
- Post—/post
- Contact Us—/contact

About、Posts、Post 和 Contact Us 将使用相同的布局(Content 组件)并在其中渲染。下面是最初的 React 路由代码(不是完整的最终版本):

```
<Router>
  <Route path="/" component={Content} >
    <Route path="/about" component={About} />
    <Route path="/about/company" .../>
    <Route path="/about/author" .../>
    <Route path="/posts" component={Posts} />
    <Route path="/posts/:id" component={Post}/>
    <Route path="/contact" component={Contact} />
  </Route>
</Router>
```

有趣的是, 可以嵌套路由以复用父组件的布局, 而且它们的 URL 可以独立于嵌套。例如, 可以将使用 /about 作为 URL 的 About 组件嵌套, 虽然父组件 Content 的布局使用的是 /app。About 组件仍将具有 Content 组件的布局(由 Content 组件的 this.props.children 实现):

```
<Router>
  <Route path="/app" component={Content} >
    <Route path="/about" component={About} />
    ...
  </Route>
</Router>
```

换句话说, About 组件不需要嵌套的 URL/app/about, 除非希望如此。这使路径和布局更具灵活性。

为了导航, 需要实现图 13.2 所示的菜单。菜单和标题将由 Content 组件渲染, 并在 About、Posts、Post 和 Contact Us 页面中复用。在图 13.2 中, 有几件事情正在发生: About 页面被渲染完, 菜单按钮处于活动状态, URL 通过显示/#/about 反映处在 About 页面上,

而文本 Node.University 反映了 About 组件中的内容(稍后将看到)。

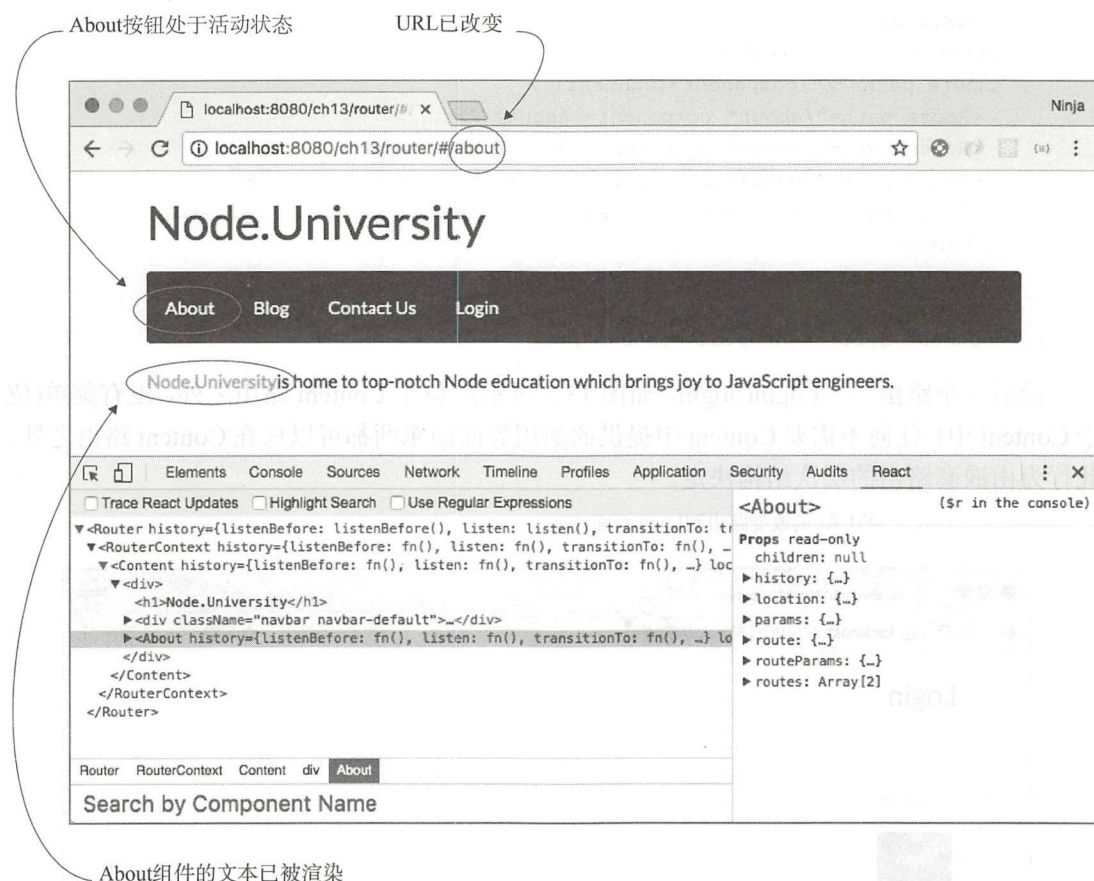


图 13.2 导航到/about 后, 在 Content 组件中渲染 About 组件的文本, 更改 URL 并激活按钮

### 13.2.1 React Router 的 JSX 样式

如前所述, 将使用 JSX 创建 Router 元素以及嵌套在其中(以及彼此)的 Route 元素。每个元素(Router 或 Route)至少有两个属性—`path` 和 `component`, 用来告诉路由 URL 路径和将要创建和渲染的 React 组件类。可以通过自定义属性来传递数据; 也可以使用自定义属性传递 `posts` 数组。

下面学以致用, 导入 React Router 对象并在 `ReactDOM.render()` 中使用它们来定义路由行为, 如代码清单 13.5 所示(ch13/router/jsx/app)。除了 About、Posts、Post 和 Contact Us 页面之外, 还将创建 Login 页面。

#### 代码清单13.5 定义Router

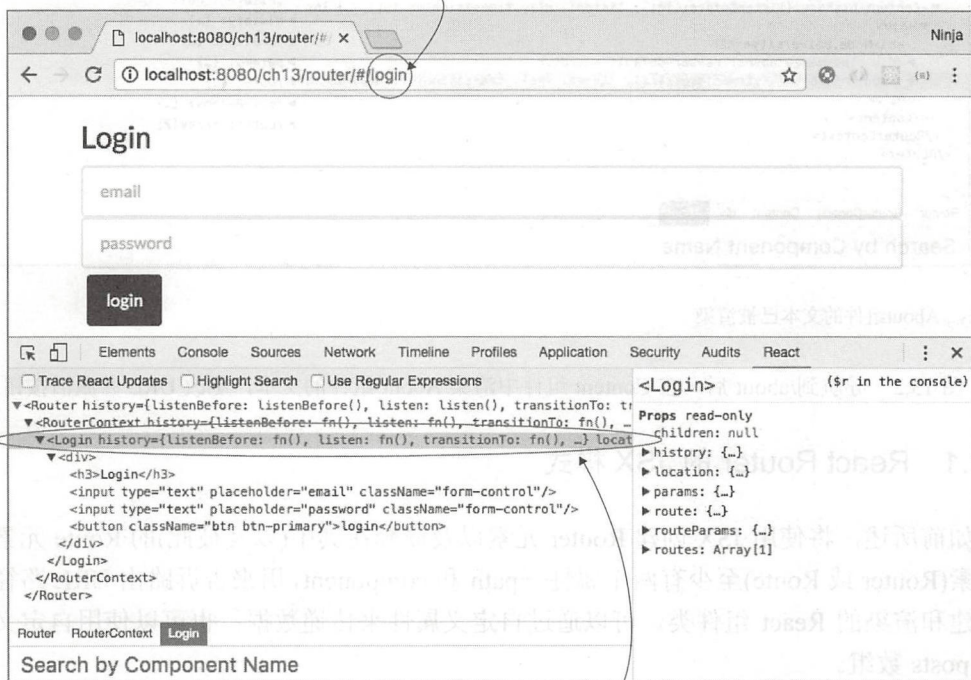
```
const ReactRouter = require('react-router')
let { Router,
    Route,
    Link
```

```
} = ReactRouter

ReactDOM.render((
  <Router history={hashHistory}>
    <Route path="/" component={Content} >
      <Route path="/about" component={About} />
      <Route path="/posts" component={Posts} posts={posts}/>
      <Route path="/posts/:id" component={Post} posts={posts}/>
      <Route path="/contact" component={Contact} />
    </Route>
    <Route path="/login" component={Login}/>
  </Router>
), document.getElementById('content'))
```

最后一个路由——Login(/login, 如图 13.3 所示), 位于 Content 路由之外, 没有菜单(位于 Content 中)。任何不需要 Content 中提供的通用界面的东西都可以放在 Content 路由之外。此行为由嵌套路由的层次结构决定。

将URL的改变路由到Login页面



Login位于Content路由之外

图 13.3 Login 页面(/#login)没有使用包含菜单的通用路由布局(Content), 只有一个 Login 元素

Post 组件根据传递的 slug(URL 的一部分, 假设是 id)渲染博客帖子, 它们可通过 props.params.id 变量从 URL(例如/posts/http2)获取。通过在路径中使用带冒号的特殊语法, 可以通知路由解析该值并将其填充到 props.params 中。

Router 被传递到 ReactDOM.render() 方法。注意将 history 传递给 Router。从 React Router 2 开始, 必须提供 history 的实现。有两个选择: 与 React Router 的 history 绑定或使用独立



的 history 实现。

### 13.2.2 哈希记录

如你所料, 哈希记录依赖哈希符号#, 这是导航页面而不重新加载的原因; 例如, 路由 `/#/posts/http2`。大多数 SPA 应用使用哈希, 因为它们需要反映应用中上下文的更改, 而不会导致完全刷新(向服务器请求)。我们在从零开始实现路由的时候就在这样做了。

注意: 哈希的正确术语是片段标识符([https://en.wikipedia.org/wiki/Fragment\\_identifier](https://en.wikipedia.org/wiki/Fragment_identifier))。

在这个例子中, 也可以使用哈希记录, 它们是从 history 库(<http://npmjs.org/history>)中独立出来的。需要导入此库, 初始化它并将它传递给 React Router。

当初始化 history 时, 需要将 `queryKey` 设置为 `false`, 因为需要禁用麻烦的查询字符串(例如 `?_k=vl8reh`), 默认支持旧版浏览器并在导航时传递状态:

```
const ReactRouter = require('react-router')
const History = require('history')
let hashHistory = ReactRouter.useRouterHistory(History.createHashHistory) ({
  queryKey: false
})
<Router history={hashHistory}/>
```

像下面这样从 React Router 导入打包完的哈希记录:

```
const { hashHistory } = require('react-router')
<Router history={hashHistory} />
```

如果需要, 可以将 React Router 与不同的 history 实现共同使用。旧版浏览器对哈希记录支持较好, 但是那意味着将会看到哈希符号#。如果需要 URL 不包括哈希标记, 也可以实现。只需要切换到浏览器记录并完成一些服务器修改。如果使用 Node 作为 HTTP 后台服务器, 这些都很简单。为保持项目的简单性, 将会使用哈希记录, 但是我们会简要介绍一下浏览器记录。

### 13.2.3 浏览器记录

一种可选的替代哈希记录的是浏览器的 HTML5 `pushState` 记录。例如, 浏览器记录模式的 URL 可能是路由 `/posts/http2` 而不是 `/#/posts/http2`。浏览器记录模式的 URL 也叫真实 URL。

浏览器记录使用常规的、无片段的 URL, 因此每个请求都会触发一个服务器请求。这就是这种方式需要服务器端配置的原因, 在此不再涉及。典型的, SPA 需要使用片段/哈希式的 URL, 特别是如果需要在旧版浏览器的话, 因为浏览器记录需要更复杂的实现。

可以像使用哈希记录一样使用浏览器记录。导入模块、插入模块, 最后配置服务器, 使之提供相同的文件(而不是 SPA 路由中的文件)。

浏览器实现来自独立的自定义包(如 history)或来自 React Router(`ReactRouter.BrowserHistory`)中的实现。导入浏览器记录库后, 将之应用到 Router:

```
const { browserHistory } = require('react-router')
<Router history={browserHistory} />
```

接下来，需要修改服务器，以便使用相同的文件响应路由 URL，而不管 URL 是什么。本例只是一种可能的实现；它使用 Node.js 和 Express：

```
const express = require('express')
const path = require('path')
const port = process.env.PORT || 8080
const app = express()

app.use(express.static(__dirname + '/public'))

app.get('*', function (request, response) {
  response.sendFile(path.resolve(__dirname, 'public', 'index.html'))
})

app.listen(port)
console.log("server started on port " + port)
```

HTTP 服务器要求服务器端行为的原因是，一旦切换到没有哈希符号的真实 URL，它们就会开始向服务器发送请求。服务器需要为每个请求提供相同的 SPA JavaScript 代码。例如，对/posts/57b0ed12fa81dea5362e5e98 和/about 的请求应在 index.html 中解析，而不是在 57b0ed12fa81dea5362e5e98.html 或 about.html 中解析(可能会导致“404: 未找到”错误)。

由于当需要支持旧版浏览器时，哈希记录可以以更优雅的方式实现 URL 路由，并且不需要配置后端服务器就可以保持例子简单，因此在本章中将使用哈希记录。

### 13.2.4 使用 Webpack 安装 React Router 开发环境

当使用 React Router 时，有一些需要使用或导入的库，和 JSX 编译器一样，需要运行它们。来看一下使用 Webpack 的 React Router 的开发和配置，它将执行这些任务。

下面的代码清单 13.6 显示了 package.json(/posts/ch13/router/package.json)中的 dev Dependencies。对于这里的大部分内容你应该已经很熟悉了。新的包有 history 和 react-router。

一如既往，确保明确使用了显示的版本；否则，不能保证代码能够运行。

代码清单13.6 依赖使用了Webpack v1、React Router v2.6、React v15.2和JSX

```
{
  ...
  "devDependencies": {
    "babel-core": "6.11.4",
    "babel-loader": "6.2.4",
    "babel-preset-react": "6.5.0",
    "history": "2.1.2",
    "react": "15.2.1",
    "react-dom": "15.2.1",
    "react-router": "2.6.0",
    "webpack": "1.12.9"
  }
}
```

除 devDependencies 外，package.json 还必须有 babel 配置。建议添加如下 npm 脚本：

```

{
  ...
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1",
    "build": "./node_modules/.bin/webpack -w",
    "i": "rm -rf ./node_modules && npm cache clean && npm install"
  },
  "babel": {
    "presets": [
      "react"
    ]
  },
  ...
}

```

请注意，因为 JSX 即将被转换为 `React.createElement()`，所以即使不使用 React，也需要在使用 JSX 的文件中导入和定义 React。为了加以说明，在代码清单 13.7 中，它看起来像没有使用 React 的 About 组件(它是无状态的，即一个函数)。但是当这段代码被转译后，它将以 `React.createElement()` 调用的形式使用 React。在第 1 章和第 2 章中，React 被定义为全局变量 `window.React`；若采用模块化而非全局的方式，则并非如此。因此，需要显式定义 `React(ch13/router/jsx/about.jsx)`。

#### 代码清单13.7 显式定义React

```

const React = require('react')

module.exports = function About() {
  return <div>
    <a href="http://Node.University" target="_blank">Node.University</a>
    is home to top-notch Node education which brings joy to JavaScript
    ➡ engineers.
  </div>
}

```

其余的文件和整个项目将使用下面这种结构：

```

/router
  /css
    bootstrap.css
    main.css
  /js
    bundle.js
    bundle.js.map
  /jsx
    about.jsx
    app.jsx
    contact.jsx
    content.jsx
    login.jsx
    post.jsx
    posts.jsx
  /node_modules
    index.html
    package.json
    posts.js
    webpack.config.js

```

打包完的文件及其源码映射，以便更好地进行调试

博客帖子的数据，例如 URL、标题和文本



index.html 文件是骨架，因为它只包含打包后的文件，如代码清单 13.8 所示。

代码清单13.8 Index.html

```
<!DOCTYPE html>
<html>

  <head>
    <link href="css/bootstrap.css" type="text/css" rel="stylesheet"/>
    <link href="css/main.css" type="text/css" rel="stylesheet"/>
  </head>

  <body>
    <div id="content" class="container"></div>
    <script src="js/bundle.js"></script>
  </body>

</html>
```

webpack.config.js 至少需要入口点 app.jsx、babel-loader 和源码映射，如代码清单 13.9 所示 (ch13/router/ webpack.config.js)。

代码清单13.9 配置Webpack

```
module.exports = {
  entry: './jsx/app.jsx',
  output: {
    path: __dirname + '/js/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',
  stats: {
    colors: true,
    reasons: true
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
}
```

设置devtool的值以查看到JSX  
源码而不是转译后的代码映射

接下来，让我们实现 Content 布局组件。

### 13.2.5 创建布局组件

Content 组件被定义为父路由，将会用于 About、Posts、Post 和 Contact 组件的布局。图 13.4 展示了它是如何实现的。

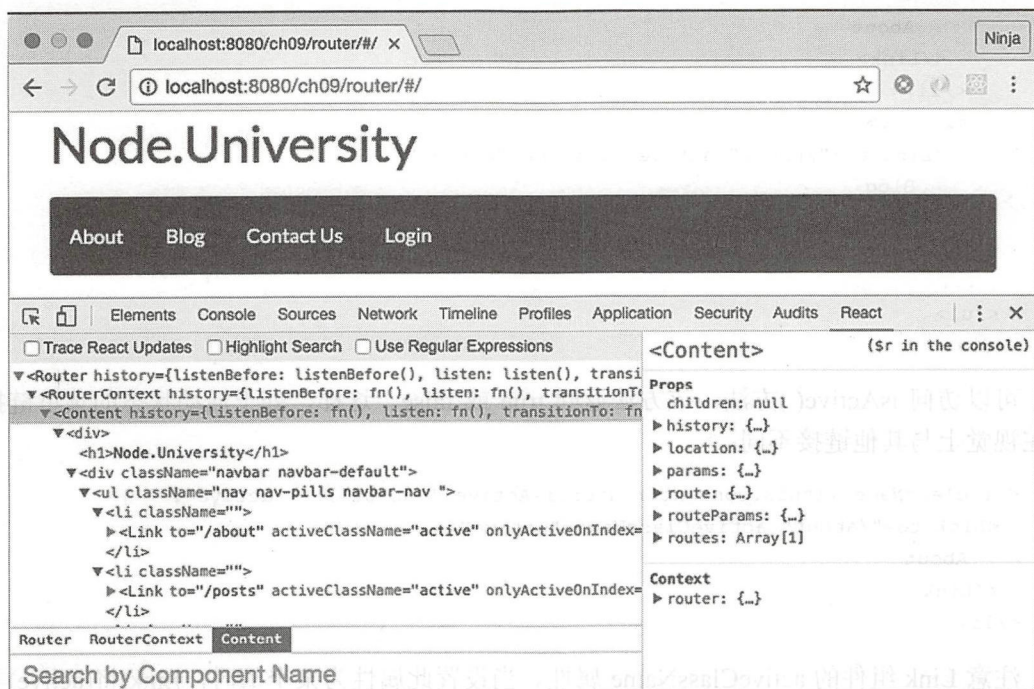


图 13.4 Content 组件作 Home 页面(没有子节点)

首先, 需要从 React Router 导入 React 和 Link。后者是渲染导航链接的特殊组件。Link 是 `<a>` 标签的特殊封装, 拥有普通 `<a>` 标签所没有的一些神奇属性, 例如 `activeClassName="active"`。当路由处于活动状态时, 该属性添加了 active 类。

Content 组件的结构看起来像下面这样, 此处省略了部分片段(后面会展示完整的代码):

```
const React = require('react')
const {Link} = require('react-router')

class Content extends React.Component {
  render() {
    return (
      <div>
        ...
      </div>
    )
  }
}

...
module.exports = Content
```

在 `render()` 中, 使用神奇的 Twitter Bootstrap UI 库(<http://getbootstrap.com>)中合适的类来声明菜单。菜单可以使用已有的 CSS 类来创建, 比如下面这些:

```
<div className="navbar navbar-default">
  <ul className="nav nav-pills navbar-nav">
    <li ...>
      <Link to="/about" activeClassName="active">
```

```

      About
    </Link>
  </li>
  <li ...>
    <Link to="/posts" activeClassName="active">
      Blog
    </Link>
  </li>
  ...
</ul>
</div>

```

可以访问 `isActive()` 方法, 该方法返回 `true` 或 `false`。这样, 处于活动状态的菜单链接将在视觉上与其他链接不同:

```

<li className={ (this.context.router.isActive('/about')) ? 'active': '' }>
  <Link to="/about" activeClassName="active">
    About
  </Link>
</li>

```

注意 `Link` 组件的 `activeClassName` 属性。当设置此属性为某个值时, `Link` 将 `(active)` 类应用于活动的元素(选中的链接)。但是需要设置 `<li>` 的样式, 而不仅仅是在 `Link` 上设置。这就是为什么也使用 `router.isActive()` 的原因。

在完成 `Content` 类定义(即将完整实现)后, 可以定义静态字段/属性 `contextTypes`, 以启用 `this.context.router`。如果使用 ES2017+/ES8+<sup>2</sup>, 则可能支持静态字段, 但在 ES2015/ES6 和 ES2016/ES7 中不是这样。它们不支持此特性。但在撰写本书时, ES2017/ES8 标准还没有最终确定, 因此也没有这个特性。

确保检查当前列表的已完功能/特性<sup>3</sup>, 或考虑使用 ES 下阶段的提案(第 0 阶段提案的收集)。

如果需要的话, `contextTypes` 静态属性将被 `React Router` 使用, `React Router` 会填充 `this.context`(从而可以访问 `router.isActive()` 和其他方法):

```

Content.contextTypes = {
  router: React.PropTypes.object.isRequired
}

```

将 `contextType` 和 `router` 设置为 `required`, 以便可以访问 `this.context.router.isActive('/about')`, 当路由处于活动状态时, 会反向通知。

代码清单 13.10 中是 `Content` 布局的完整实现。

2 通过 <https://node.university/blog/498412/es7-es8> 和 <https://node.university/p/es7-es8> 可了解 ES2016/ES7 和 ES2017/ES8 功能的更多信息

3 有关第 0 至第 3 阶段已完成提案的最新列表, 请参阅 Github 上的 TC39 文档: <https://github.com/tc39/proposals/blob/master/README.md> 和 <https://github.com/tc39/proposals/blob/master/finished-proposals.md>



代码清单13.10 完整的Content组件

```

const React = require('react')
const {Link} = require('react-router')

class Content extends React.Component {
  render() {
    return (
      <div>
        <h1>Node.University</h1>
        <div className="navbar navbar-default">
          <ul className="nav nav-pills navbar-nav">
            <li className={ (this.context.router.isActive('/about')) ?
              ➡ 'active': '' }>
              <Link to="/about" activeClassName="active">
                About
              </Link>
            </li>
            <li className={ (this.context.router.isActive('/posts')) ?
              ➡ 'active': '' }>
              <Link to="/posts" activeClassName="active">
                Blog
              </Link>
            </li>
            <li className={ (this.context.router.isActive('/contact')) ?
              ➡ 'active': '' }>
              <Link to="/contact" activeClassName="active">
                Contact Us
              </Link>
            </li>
            <li>
              ➡ 使用Link来创建导航链接
              <Link to="/login" activeClassName="active">
                Login
              </Link>
            </li>
          </ul>
        </div>
        {this.props.children}
      </div>
    )
  }
  ➡ 定义这个组件需要上下文中的路由对象
}
Content.contextTypes = {
  router: React.PropTypes.object.isRequired
}
module.exports = Content

```

访问Router及其方法来检查活动的路由

渲染子路由(在app.jsx中定义)

`children` 语句使之可以在每个子路由(嵌套在/路由中的路由)上复用菜单, 例如/`posts`、/`post`、/`about` 和/`contact`:

```
{this.props.children}
```

除了使用 `contextTypes` 外, 让我们来看看另一种访问单个路由中路由器的方式。

## 13.3 React Router 特性

为了解有关 React Router 的功能和模式的更多信息, 下面看一下从子组件访问路由器

的另一种方法，以及在这些组件中如何以编程方式导航。当然，如果不介绍如何解析 URL 参数和传递数据，那么这一章就不算完整。

### 13.3.1 使用 withRouter 高阶组件访问路由器

使用路由器允许以编程的方式导航并访问当前路由。在组件中包含对路由器的访问是很好的。

你已经了解到如何通过设置静态类属性从 `this.context.router` 访问路由器：

```
Content.contextTypes = {  
  router: React.PropTypes.object.isRequired  
}
```

在某种程度上，这是在使用验证机制定义 API；也就是说，组件必须有路由器。Content 组件使用了这种方法。

但是 Content 组件依赖 React 的上下文环境，这是一种实验性方式。React 团队并不建议使用。幸运的是，还有另一种方式(有些人可能会认为这种方式更简单、更好；请参阅 <http://mng.bz/Xhb9>)：withRouter。

withRouter 是高阶组件(HOC，关于高阶组件，详见第 8 章)，它以组件为参数，注入路由器，并返回另一个 HOC。例如，可以像下面这样将路由器注入 Contact 组件：

```
const {withRouter} = require('react-router')  
...  
<Router ...>  
  ...  
  <Route path="/contact" component={withRouter(Contact)} />  
</Router>
```

当查看 Contact 组件的实现(一个函数)时，路由器对象可以从属性(函数的参数对象)访问到：

```
const React = require('react')  
module.exports = function Contact(props) {  
  // props.router - GOOD!  
  return <div>  
    ...  
  </div>  
}
```

withRouter 的优点是可以与常规的有状态 React 类以及无状态函数一起工作。

注意：即使没有直接(可见地)使用 React，也必须引入 React，因为这些语法会被 `React.createElement()` 语句转换为代码，这些语句依赖 React 对象。更多相关信息，请参阅第 3 章。

### 13.3.2 以编程方式导航

路由器的另一种流行用法是以编程方式导航：从基于逻辑的代码中更改 URL(位置)，而不是基于用户行为更改。为了演示，假设有一个应用，用户在通讯录表单上键入一条信息，然后提交表单。根据服务器响应，应用导航到 Error 页面、Thank-you 页面或 About 页面。

一旦有了路由器，就可以根据需要，通过调用 `router.push(URL)` 用编程方式导航，如代码清单 13.11 所示。这里的 URL 必须是已定义的路由路径。例如，可以在 1 秒种后从 Contact 页面导航到 About 页面。

代码清单 13.11 调用 `router.push()` 进行导航

```
const React = require('react')

module.exports = function Contact(props) {
  setTimeout(() => {props.router.push('about')}, 1000)
  return <div>
    <h3>Contact Us</h3>
    <input type="text" placeholder="your email" className="form-control" />
    <textarea type="text" placeholder="your message" className="form-control" />
    <button className="btn btn-primary">send</button>
  </div>
}
```

在1秒种后导航离开

以编程方式导航是一项重要的功能，因为这可以更改应用的状态。来看看如何访问诸如帖子 id 之类的 URL 参数。

### 13.3.3 URL 参数和其他路由数据

由上可知，使用 `contextTypes` 和 `router` 会获得 `this.context.router` 对象。它是 `app.jsx` 中定义的 `<Router>` 的一个实例，可用于导航、获取活动路径等。另一方面，`this.props` 中还有其他有趣的信息，不需要使用一些静态属性来访问它：

- `history(v2.x 版本已弃用，可以改用 context.router)`
- `location`
- `params`
- `route`
- `routeParams`
- `routes`

`this.props.location` 和 `this.props.params` 对象包含当前路由的数据，如路径名称、URL 参数(名字使用冒号定义)等。

在 `Post` 组件所在的 `post.jsx` 文件的 `Array.find()` 中使用 `params.id` 来查找与 URL 路径相一致的帖子，例如路由 `/#/posts/http2(ch13/router/jsx/post.jsx)`，参见代码清单 13.12。



代码清单 13.12 渲染帖子数据

```
const React = require('react')

module.exports = function Product(props) {
  let post = props.route.posts.find(element=>element.slug ==
    props.params.id)
  return (
    <div>
      <h3>{post.title}</h3>
      <p>{post.text}</p>
      <p><a href={post.link} target="_blank">Continue reading...</a></p>
    </div>
  )
}
```

通过slug属性找到一个帖子

当导航到 Posts 页面(参见图 13.5)时, 有一个帖子列表。稍作提示, 路由的定义如下:

```
<Route path="/posts" component={Posts} posts={posts}/>
```

单击一个帖子, 导航到#/posts/ID。该页面复用了 Content 组件的布局。

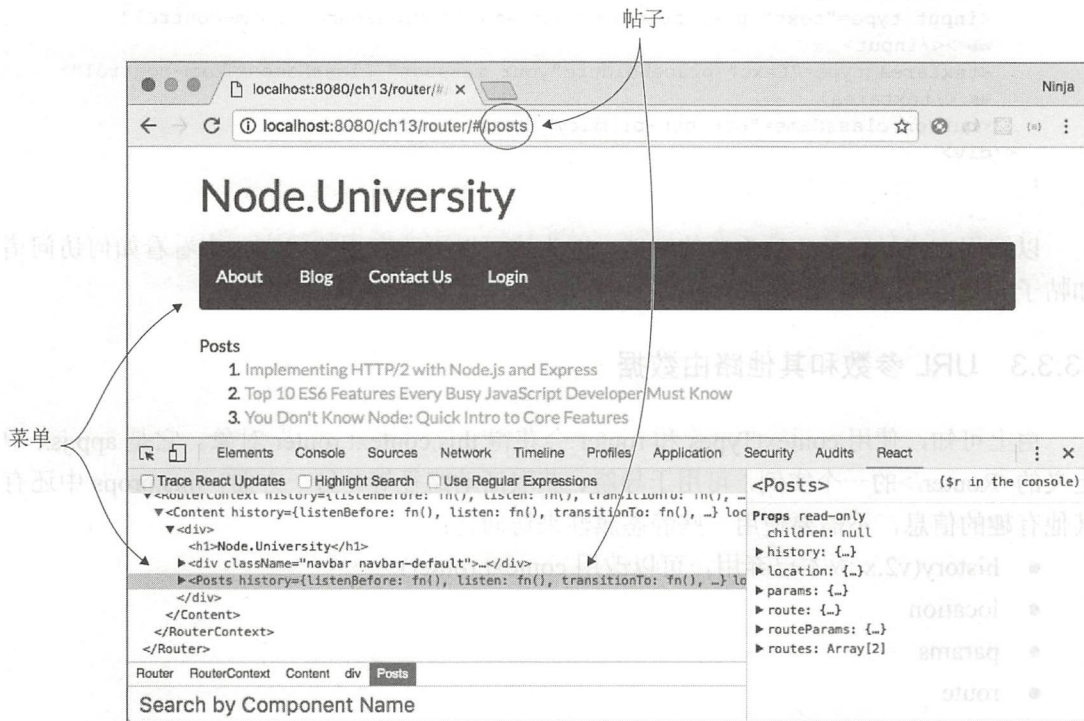


图 13.5 Posts 页面在 Content(菜单)组件中渲染 Posts 组件, 因为在 app.jsx 中,

Posts 是作为 Content 的子路由定义的

现在, 让我们继续处理数据。

### 13.3.4 在 React Router 中传递属性

经常需要将数据传递给嵌套的路由, 而且很容易做到。在例子中, Posts 需要获取关于

帖子的数据。在代码清单 13.13 中, `post.js` 文件中的 `Posts` 可以访问在 `app.jsx` 文件的 `<Route/>` 中传递的 `posts` 属性。可以将任何数据作为属性传递给路由, 例如 `<Route path="/posts" component={Posts} posts={posts}/>`。然后可以访问 `props.route` 中的数据, 例如 `props.route.posts` 是一个帖子列表。

代码清单13.13 Posts使用props.route中的数据来实现

```
const {Link} = require('react-router')
const React = require('react')

module.exports = function Posts(props) {
  return <div>Posts
    <ol>
      {props.route.posts.map((post, index)=>
        <li key={post.slug}><Link
```

访问在路由  
声明中定义  
的属性

当然, 此数据的值可以是一个函数。这样可以像 `app.jsx` 一样, 将事件处理程序传递给无状态组件, 并且仅在主组件中实现这些程序。

现在已经完成了所有主要的部分, 并准备好了启动这个项目! 可以通过运行 `npm` 脚本 (`npm run build`)或直接使用 `./node_modules/.bin/webpack -w` 来完成。等待构建完成后, 你会看到像下面这样的情况:

```
> router@1.0.0 build /Users/azat/Documents/Code/react-quickly/ch13/router
> webpack -w
```

```
Hash: 07dc6eca0c3210dec8aa
```

```
Version: webpack 1.12.9
```

```
Time: 2596ms
```

Asset	Size	Chunks	Chunk Names
bundle.js	976 kB	0 [emitted]	main
bundle.js.map	1.14 MB	0 [emitted]	main
+ 264 hidden modules			

在新的窗口中, 打开你最喜欢的静态服务器(可以使用 `node-static`, 也可以使用 `Express` 创建自己的静态服务器), 并导航到浏览器中的位置。尝试导航到 `/` 和 `/#/about`, 具体的 URL 将取决于在同一文件夹还是在父文件夹中运行的静态服务器。

注意: 由于篇幅受理, 这里不包括例子的完整代码。如果想运行它或使用它作为样板, 或当脱离上下文时, 你发现之前的代码片段比较混乱, 可以在 [www.manning.com/books/react-quickly](http://www.manning.com/books/react-quickly) 或 <https://github.com/azat-co/react-quickly/tree/master/ch13/router> 上找到本例完整的代码。

## 13.4 使用 Backbone 路由

当需要对单页面应用进行路由时, 可以直接将 `React` 与其他路由或类 `MVC` 库一起使用。例如, `Backbone` 是内置了前端 URL 路由的最流行的前端框架之一。让我们来看看如何通过执行以下操作来轻松使用 `Backbone` 路由渲染 `React` 组件:

- 使用 `routes` 对象定义一个类作为从 URL 片段到功能的映射。
- 在 `Backbone Router` 类的方法/函数中渲染 `React` 元素。
- 实例化并启动 `Backbone Router` 对象。

项目结构如下：

```
/backbone-router
/css
  bootstrap.css
  main.css
/js
  bundle.js
  bundle.map.js
/jsx
  about.jsx
  app.jsx
  contact.jsx
  content.jsx
  login.jsx
  post.jsx
  posts.jsx
/node_modules
...
index.html
package.json
posts.js
webpack.config.js
```

`package.json` 除了包含 `Webpack v2.4.1`、`React v15.5.4` 和 `Babel v6.11` 之外，还包括 `Backbone v1.3.3`：

```
{
  "name": "backbone-router",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "./node_modules/.bin/webpack -w",
    "i": "rm -rf ./node_modules && npm cache clean && npm install"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"
    ]
  },
  "devDependencies": {
    "babel-core": "6.11.4",
    "babel-loader": "6.4.1",
```



```

    "babel-preset-react": "6.5.0",
    "backbone": "1.3.3",
    "jquery": "3.1.0",
    "react": "15.5.4",
    "react-dom": "15.5.4",
    "webpack": "2.4.1"
  }
}

```

主要逻辑的源代码在 `app.jsx` 中，在那里执行所有上述三项任务：

```

const Backbone = require ('backbone')
// Include other libraries
const Router = Backbone.Router.extend({
  routes: {
    '' : 'index',
    'about' : 'about',
    'posts' : 'posts',
    'posts/:id' : 'post',
    'contact' : 'contact',
    'login': 'login'
  },
  ...
})

```

一旦 `routes` 对象被定义，就可以定义这些方法。`routes` 中的值必须是方法名：

```

// Include libraries
const Router = Backbone.Router.extend({
  routes: {
    '' : 'index',
    'about' : 'about',
    'posts' : 'posts',
    'posts/:id' : 'post',
    'contact' : 'contact',
    'login': 'login'
  },
  index: function() {
    ...
  },
  about: function() {
    ...
  }
  ...
})

```

将每个 URL 片段映射到一个函数。例如，`#/about` 将会触发 `about()`。因此，可以定义这些方法并在其中渲染 React 组件。数据将会作为属性(`router` 或 `posts`)传递：

```

const {render} = require ('react-dom')
// ...
const Router = Backbone.Router.extend({
  routes: {
    ...
  },
  index: function() {
    render(<Content router={router}/>, content)
  },
  about: function() {
    render(<Content router={router}>
      <About/>
    </Content>, content)
  },
  posts: function() {
    render(<Content>
      <Posts posts={posts}/>
    </Content>, content)
  },
  post: function(id) {
    render(<Content>
      <Post id={id} posts={posts}/>
    </Content>, content)
  },
  contact: function() {
    render(<Content>
      <Contact />
    </Content>, content)
  },
  login: function() {
    render(<Login />, content)
  }
})

let router = new Router()
Backbone.history.start()

```

使用解构赋值来从 ReactDOM.render() 中导入和定义 render()

创建 Content, 并且 About 在其中, 可以将路由器作为属性传递

给 Post 传递需要的数据, 例如 URL 参数(id)和帖子数据

在 Content 外渲染 Login

实例化 Router, 并启动浏览器记录

content 变量是一个 DOM 节点(在路由器之前声明):

```
let content = document.getElementById('content')
```

与 React Router 示例相比, 像 Post 这样的嵌套组件不是从 props.params 或 props.route.posts 中获取它们的数据, 而是从 props.id 和 props.posts 中获取。另一方面, 不能使用声明式的 JSX 语法, 而必须使用强制风格。

此项目的完整代码可以从 [www.manning.com/books/reactquickly](http://www.manning.com/books/reactquickly) 和 <https://github.com/azat-co/react-quickly/tree/master/ch13/backbonerouter> 上获取到。如果有一个 Backbone 系统或者打算使用 Backbone, 这个例子将会给你提供一个开端。即使没有计划使用 Backbone, 它也再一次显示了 React 与其他库共用时的惊人之处。

## 13.5 测验

1. 必须为 React Router v2.x 提供一个 history 实现, 因为默认情况下不会使用 history。

这么理解正确还是错误？

2. 什么样的 history 实现对旧版本浏览器支持更好：哈希记录还是浏览器的 HTML5 pushState 记录？
3. 当使用 React Router v2.x 时，需要实现什么来访问路由组件中的 router 对象？
4. 当使用 React Router v2.x 时，在路由组件(无状态或有状态组件)中如何访问 URL 参数？
5. React Router 需要使用 Babel 和 Webpack，这么理解正确还是错误？

## 13.6 小结

- 可以通过监听哈希的改变，使用 React 以一种纯朴的方式实现路由。
- React Router 提供用于定义路由层次结构的 JSX 语法：`<Router><Route/></Router>`。
- 嵌套路由不需要相对于父路由的嵌套 URL，路径和嵌套是独立的。
- 通过将 queryKey 设置为 false，可以使用没有符号标记的哈希路由。
- 当使用 JSX 时，即使没有明显使用 React，也必须包含 `React(require('react'))`，因为将 JSX 转换为 `React.createElement()` 需要 React。

## 13.7 测验答案

1. 正确。React Router v1.x 默认加载一个 history 实现；但是在版本 2.x 中，必须提供一个库，可以来自单独的包或使用路由器库打完的包。
2. 哈希记录对旧版本浏览器支持更好。
3. 与 router 一起的静态类属性 contextTypes 是必需的对象。
4. 从 `props.params` 或 `props.routeParams` 访问。
5. 错误。可以简单地使用 React Router 和/或其他构建工具，如 Gulp 和 Browserify。



# 第 14 章

## 使用 Redux 处理数据

本章内容:

- 了解 React 中的单向数据流
- 了解 Flux 数据体系结构
- 使用 Redux 处理数据

到目前为止，一直在使用 React 创建用户界面。这是 React 最常见的用例。但是大多数 UI 都需要处理数据。这些数据来自服务器(后端)或其他浏览器组件。

在处理数据方面，React 提供了很多选择:

- 与类 MVC 框架结合——如果已经在使用或正在计划为单页面应用使用类 MVC 框架，那么这种方案非常理想:例如，使用 Backbone 和 Backbone 模型。
- 编写自己的数据方法或库——这种方案非常适合小型 UI 组件:例如，为 UI 中的账户表格获取账户列表。
- 使用 React 技术栈(又称 React 全家桶)——这种方案提供最大的兼容性(代码整合会有更少的冲突)和最完整的 React 哲学。

本章介绍第三种方案中最受欢迎的选择之一: Redux。首先概述数据在 React 组件中是如何流动的。

注意: 还有 Flux 框架和来自 Facebook 的 flux 库。本章将向你展示 Redux 而不是 flux 库，因为 Redux 在项目中更常用。flux 试验了 Flux 框架的概念，而这也正是 React 所要坚持并实现的。你可以将 Redux 和 flux(库)看成 Flux 架构的两个实现(本章将涵盖 Flux 架构但不是 flux 库)。

## 14.1 React 支持单向数据流

React 是设计用来处理单向数据流的视图层(见图 14.1)。当所关注的视图之间不存在可变(或双向)引用时,则存在单向数据流模式(也称为单向绑定)。关注的视图是指具有不同功能的部分。例如,视图和模型不能有双向引用。一会儿再谈双向数据流。

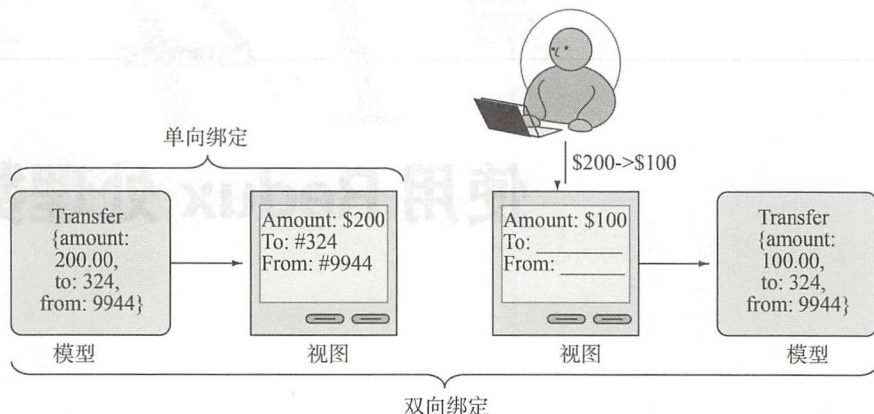


图 14.1 单向与双向数据流的对比

为了说明,如果有账户模型和账户视图,那么数据只能从账户模型流到账户视图,而不能反向流动。换句话说,模型的更改将会导致视图的改变(见图 14.2)。理解这一点的关键是视图不能直接修改模型。

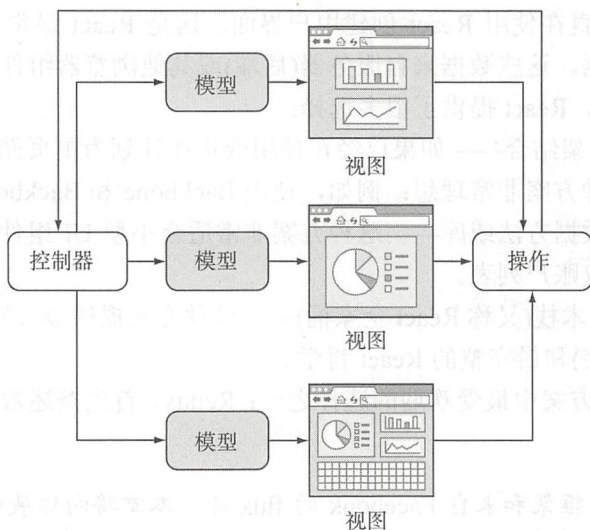


图 14.2 单向数据流的简化视图,其中视图不能直接修改模型

单向数据流可以确保对于组件来说,给定的任何输入,都将得到相同的可预测的结果:一个 `render()` 表达式。React 的这种模式与 Angular 和其他一些使用双向绑定模式的框架形成鲜明的对比。

例如,在双向数据流中,模型中的更改会导致视图的改变,并且视图中的更改(用户输

入)也会导致模型的改变。由于这个原因,在双向数据流中,视图的状态是不可预测的,这使其更难理解、调试和维护(见图 14.3)。需要记住的关键点是:视图可以直接修改模型。这与单向数据流形成鲜明的对比。

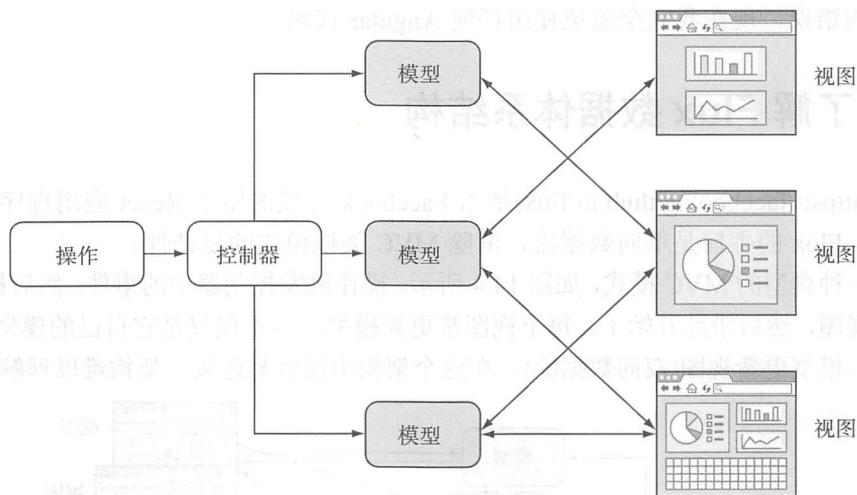


图 14.3 类 MVC 架构的典型双向数据流的简化视图

有趣的是,双向数据流(双向绑定)被一些 Angular 开发者认为是优点。为避免陷入争论,使用双向数据流确实可以减少代码量。

例如,假设有一个输入字段,如前面的图 14.1 所示。只需要在模板中定义一个变量,当用户在该字段中输入时,值将在模型中更新。同时,如果模型发生改变(例如作为 XHR GET 请求的结果),网页上的值将被更新。因此,改变是双向的:从视图到模型,再从模型到视图。这对于原型设计来说很好,但对于复杂的用户界面来说,在性能、调试、开发扩展等方面却不尽如人意。这可能听起来有争议——请见谅。

我已经构建了许多使用 MVC 和 MVW(Model-View-Whatever)框架的复杂 UI 应用,它们涉及双向数据流并且完成了双向流动的工作。简而言之,由于每个视图可以操作每个模型,问题会增加,反之亦然。当有一个或两个隔离的模型和视图时,这很好。但是应用越大,就有越多的模型和视图互相更新。要弄清楚某个模型或视图为什么处于一个给定的状态变得越来越困难,因为无法容易地确定哪个模型/视图更新了它,以及是按照什么顺序更新的。可追踪性成为一个巨大的问题,寻找错误也一样。这就是为什么 MVC 框架(如 Angular,目前业内主流观点认为,Angular 随着不断发展更加贴适于 MVVM 框架)中的双向数据流不受许多开发人员青睐的原因:他们发现这种反模式很难调试和扩展。

另一方面,在单向流动的情况下,模型更新视图,就是那样。作为额外的好处,单向数据流允许服务器端渲染,因为视图是状态的不可变函数(也就是同构 JavaScript)。

现在,请记住,单向数据流是 React 的一个主要卖点:

- 基于单一情形(状态/模型到视图)的代码可读性和可追踪性。
- 可调试的代码与时间旅行<sup>1</sup>;例如,在发生异常和错误时,向服务器发送带有历史记录的消息。

1 Dan Abramov, “Live React: Hot Reloading with Time Travel” (presentation, ReactEurope 2015), <http://mng.bz/uSxq>



- 无浏览器头的服务器端渲染，有些人称之为同构<sup>2</sup>或通用<sup>3</sup>JavaScript。

为了防止有疑问，以下是我个人的 Angular 使用经历。我很少使用 Angular 1，因为我觉得它是有缺陷的，后来我学习了 Angular 2——然后我意识到自己犯了很大的错误。我纠正了自己的错误，现在我完全避免使用任何 Angular 代码。

## 14.2 了解 Flux 数据体系结构

Flux(<https://facebook.github.io/flux>)是由 Facebook 开发的用于 React 应用程序数据流的架构模式。Flux 的主旨是单向数据流，消除 MVC 类似模式的复杂性。

考虑一种典型的 MVC 模式，如图 14.4 所示。操作触发控制器中的事件。然后根据模型，应用渲染视图，然后错乱开始了。每个视图都更新模型——不仅仅是它自己的模型，还有其他模型——模型更新视图(双向数据流)。在这个架构中很容易迷失。架构难以理解和调试。

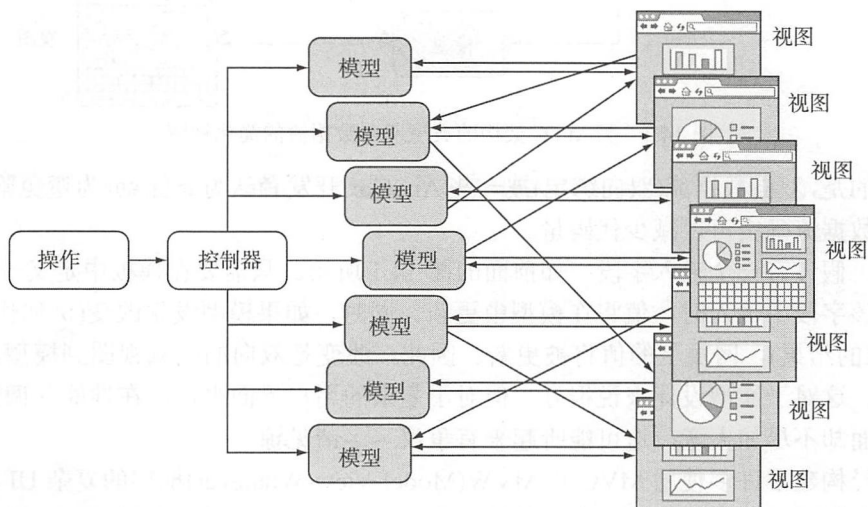


图 14.4 类 MVC 架构由于允许视图触发的更改可以作用于任何模型，从而引入了复杂性，反之亦然

相反，Flux 建议使用单向数据流，如图 14.5 所示。在这种情况下，将通过分发器处理视图的操作，分发器依次调用数据存储(Flux 是 MVC 的替代品，这不仅仅是新的术语)。数据存储负责数据和视图中的表示。视图不会修改数据，但是有可以再次通过分发器处理的操作。

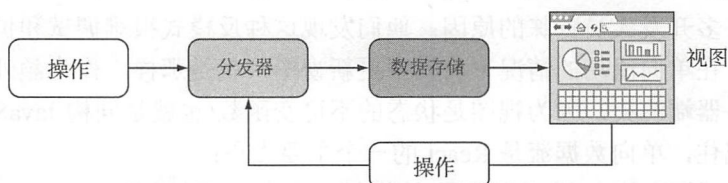


图 14.5 Flux 架构使数据单向流向(从数据存储到视图)，从而简化了数据流

2 Spike Brehm, “Isomorphic JavaScript: The Future of WebAir Apps”, Airbnb Engineering & Data Science, November 11, 2013, <http://mng.bz/i34M>

3 Michael Jackson, “Universal JavaScript”, June 8, 2015, <http://mng.bz/7GXE>

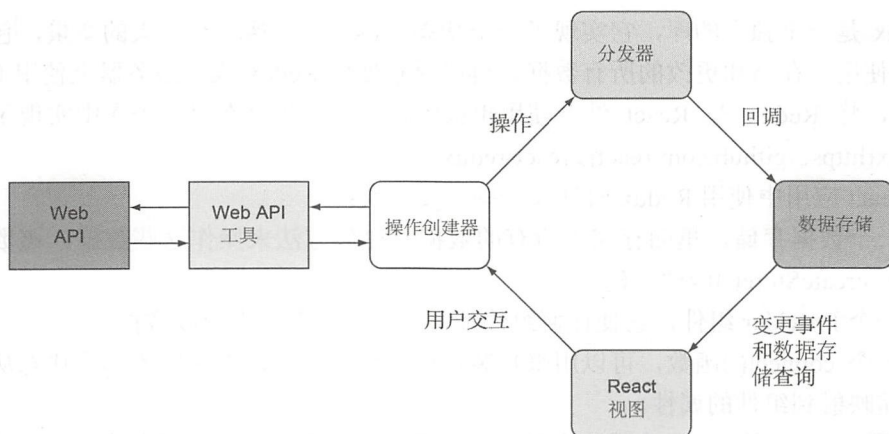


图 14.6 Flux 架构概述：操作触发分发器，分发器触发数据存储，数据存储渲染视图

单向数据流可以更好地进行测试和调试。Flux 体系结构的更详细图表如图 14.6 所示。

历史上，Flux 是一个架构。Facebook 团队后来才发布可与 React 一起使用的 flux 模块 ([www.npmjs.com/package/flux](http://www.npmjs.com/package/flux) 和 <https://github.com/facebook/flux>) 的实现。flux 模块或多或少是 Flux 架构概念的试验，React 开发者很少使用。

**提示：**此时已经不用赘述 Flux 的优秀思想了，建议观看 Flux 官网上的视频“Hacker Way: Rethinking Web App Development at Facebook”：<http://mng.bz/wygf>。

就个人而言，我觉得 Flux 有些混乱——并不是只有我一个人这样想。Flux 有很多实现，包括 Redux、Reflux 和其他库。本书的早期读者知道本书的第一版包含了 Reflux，但是在这一版中我省略了它。我的秘密证据——David Waller 的“React.js architecture - Flux vs. Reflux”（位于 <http://mng.bz/5GHx>），以及 npm 的实际下载数据，都表明 Redux 比 Flux 或 Reflux 更受欢迎。在本书中，我使用 Redux，有些人认为这是对 Flux 的更好选择。

## 14.3 使用 Redux 数据类库

Redux([www.npmjs.com/package/redux](http://www.npmjs.com/package/redux))是 Flux 架构最流行的实现之一。Redux 有如下这些特性：

- 生态繁荣，详见 Awesome Redux(<https://github.com/xgrommx/awesome-redux>)。
- 简单，不需要分发器或注册数据存储，最小版本只有 99 行代码(<http://mng.bz/00Ap>)。
- 良好的开发体验(DX)，例如，可以随时间旅行进行热加载(见视频“Live React: Hot Reloading with Time Travel”(<http://mng.bz/uSxq>))。
- Reducer 组合，例如，撤销/重复高阶组件只需要很少的代码(<http://github.com/omnidan/redux-undo>)。
- 支持服务器端渲染。

这里不会花时间去了解为什么 Redux 比 Flux 更好的细节。如果有兴趣，可以阅读 Redux 作者的一些想法：Why Use Redux over Facebook Flux?(<http://mng.bz/z9ok>)。

Redux 是一个独立的库，它实现了一个状态容器。它就像一个巨大的变量，包含应用在运行时使用、存储和更改的所有数据。可以单独使用 Redux 或在服务器上使用 Redux。如前所述，将 Redux 与 React 结合使用也很流行；这个组合在另一个库中实现了，就是 react-redux(<https://github.com/reactjs/react-redux>)。

在 React 应用中使用 Redux 时涉及一些主要模块：

- 一个数据存储，里面存储了所有的数据并提供方法来操作这些数据。该数据存储由 createStore() 函数创建。
- 一个 Provider 组件，它使任何组件都可以从数据存储中获取数据。
- 一个 connect() 函数，可以用来封装任何组件，并可以将应用的部分状态从数据存储映射到组件的属性。

回顾图 14.5 中的 Flux 体系结构图：可以看到为什么有一个数据存储。改变内部状态的唯一方法是分发操作，并且操作对应在数据存储中。

数据存储中的每一个改变都是通过操作来完成的。每个操作都会告诉应用发生了什么变化，以及应该改变哪个部分。操作也可以提供数据；这非常有用，因为应用要处理变更的数据。

由纯函数 reducer 指定数据存储中的数据更改方式。它们有 (state, action) => state 签名。换句话说，通过对当前数据存储使用操作，得到新的状态。这使得结果具有可预测性，并且有回退操作到之前状态的能力。

下面是 Todo 列表应用的 reducer 文件，其中 SET\_VISIBILITY\_FILTER 和 ADD\_TODO 是操作：

```
function todoApp(state = initialState, action) {  
  switch (action.type) {  
    case 'SET_VISIBILITY_FILTER':  
      return Object.assign({}, state, {  
        visibilityFilter: action.filter  
      })  
    case 'ADD_TODO':  
      return Object.assign({}, state, {  
        todos: [  
          ...state.todos,  
          {  
            text: action.text,  
            completed: false  
          }  
        ]  
      })  
    default:  
      return state  
  }  
}
```

定义一个操作

通过复制当前state和visibilityFilter的值，使用reducer来创建新的状态

定义ADD\_TODO action

通过复制当前state和新的TODO值text和completed，使用reducer创建新的状态作为todos数组的最后一项

定义默认回退，此时返回当前状态

Redux 应用中可能有一个或多个 reducer(或没有)。每次调用一个操作，就会调用一个 reducer。reducer 负责更改数据存储中的数据；这就是为什么需要注意它们在某些操作类型中做哪些事情的原因。



通常, `reducer` 是一个以数据存储和操作作为参数的函数。例如, 操作可以是“获取电影”(FETCH\_MOVIE), 通过使用一个 `reducer` 获取到。操作的代码描述了一个操作如何将数据存储转换为下一个数据存储(将电影添加到数据存储中)。这个 `reducer` 函数包含一条庞大的 `switch/case` 语句来处理操作。但令人惊喜的是: 有一个方便的库可以使 `reducer` 更具功能性并且更容易阅读。该库名叫 `redux-actions`, 你即将看到如何用它替代 `switch/case` 语句。

提示: Redux 的创建者 Dan Abramov(<https://github.com/gaearon>)建议在阅读 Redux 之前阅读以下内容: Why Use Redux Over Facebook Flux(<http://mng.bz/9syg>)和 What Could Be the Downsides of Using Redux Instead of Flux(<http://mng.bz/Ux9l>)。

### 14.3.1 用 Redux 创建仿照 Netflix 的应用

我们都喜欢好莱坞经典电影, 对吧? 下面来制作一个显示经典电影列表的应用: 也就是说, 一个仿照 Netflix 的应用(但只有首页, 没有流媒体或类似的东西)。该应用将显示一个电影网格(见图 14.7), 当单击某个电影的图像时, 会看到详细界面(见图 14.8)。

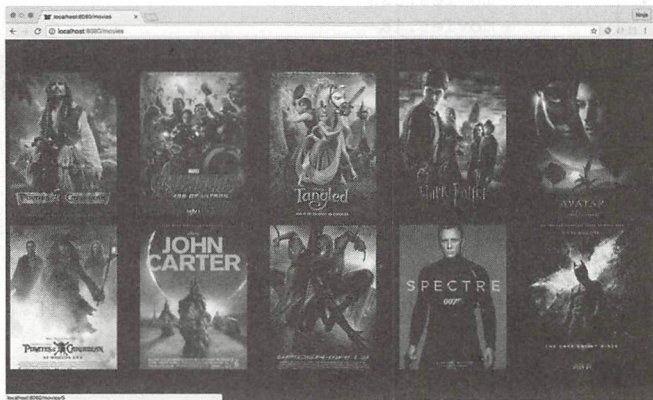


图 14.7 在首页上按照网格形式显示电影

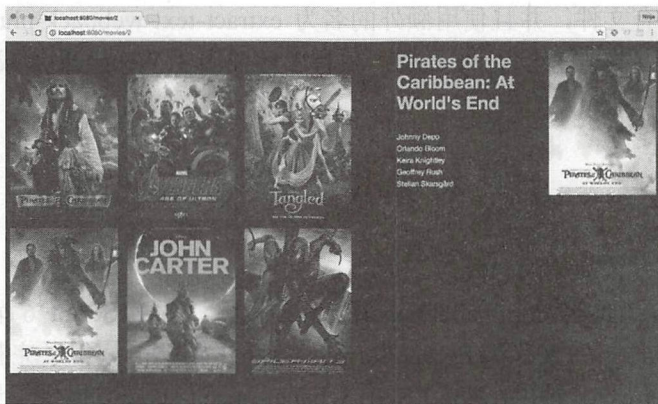
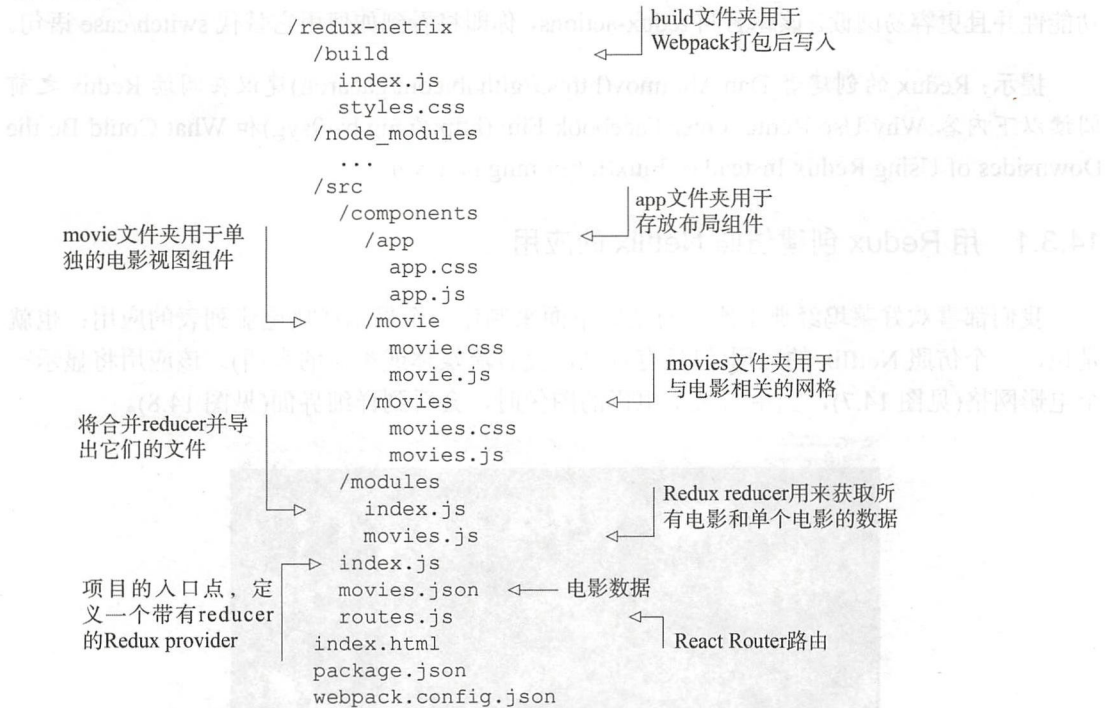


图 14.8 单击海报时会显示电影的详细信息

该项目的目标是学习如何在实际场景中使用 Redux 将数据提供给 React 组件。为保持简单, 这些数据将来自 JSON 文件。使用在前一章学到的 React Router 来处理每个电影单

独的详细视图将会更容易。

该项目将有三个组件：**App**、**Movies** 和 **Movie**。每个组件都有一个 CSS 文件，并存放于自身的文件夹中，以便更好地组织代码(这是将 React 组件与样式一起封装的最佳实践)。项目结构如下：



现在项目的文件结构已经准备就绪，来看看依赖关系和构建配置。

### 14.3.2 依赖和配置

需要为项目设置一些依赖。活学活用，将使用 Webpack(<https://github.com/webpack/webpack>)来打包所有文件，并且使用额外的名为 `extract-text-webpack-plugin` 的插件将多个 `<style>` 包含(内联)的样式打包为 `style.css`。项目中还使用了 Webpack 加载器：

- `json-loader`
- `style-loader`
- `css-loader`
- `bable-loader`

项目的其他开发依赖模块包括以下内容：

- Babel(<https://github.com/babel/babel>)及其预设将 ECMAScript 6 转换成适合浏览器的旧版 JavaScript(又称为 ECMAScript 5)：`babel-polyfill` 模拟了完整的 ES2015 环境，`babel-preset-es2015` 用于 ES6/ES2015，`babel-preset-stage-0` 提供了最新的 ES7+特性，`babel-preset-react` 用于 JSX。
- `react-router`(<https://github.com/reactjs/react-router>)基于当前位置显示组件的层次结构，还可以根据 URL 位置将组件分配到层次结构中。

- `redux-actions`(<https://github.com/acdlite/redux-actions>)组织 reducer。
- `ESLint`(<http://eslint.org>)及其插件用于保持正确的 JavaScript/JSX 风格。
- `concurrently`([www.npmjs.com/package/concurrently](http://www.npmjs.com/package/concurrently))是一个 Node 工具,可以使诸如 Webpack 构建之类的过程并行(同时)进行。

`package.json` 文件列出了所有的依赖、Babel 配置文件和 npm 脚本,并且至少应包含代码清单 14.1(ch14/redux-netflix/package.json)中显示的数据。一如既往地可以使用 `npm i NAME` 手动安装模块,输入 `package.json`,然后运行 `npm i`,或者复制 `package.json` 并运行 `npm i`。确保使用 `package.json` 中库的准确版本,否则代码可能会运行失败。

代码清单 14.1 用于Netflix克隆版的依赖

```
{
  "name": "redux-netflix",
  "version": "0.0.1",
  "description": "A sample project in React and Redux that copies Netflix's
  ➤ features and workflow",
  "main": "./build/index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "concurrently \"webpack --watch --config webpack.config.js\"
    ➤ \"webpack-dev-server\"",
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/azat-co/react-quickly.git"
  },
  "author": "Azat Mardan (http://azat.co)",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/azat-co/react-quickly/issues"
  },
  "homepage": "https://github.com/azat-co/react-quickly#readme",
  "devDependencies": {
    "babel-core": "6.11.4",
    "babel-eslint": "6.1.2",
    "babel-loader": "6.2.4",
    "babel-polyfill": "6.9.1",
    "babel-preset-es2015": "6.9.0",
    "babel-preset-react": "6.11.1",
    "babel-preset-stage-0": "6.5.0",
    "concurrently": "2.2.0",
    "css-loader": "0.23.1",
    "eslint": "3.1.1",
    "eslint-plugin-babel": "3.3.0",
    "eslint-plugin-react": "5.2.2",
    "extract-text-webpack-plugin": "1.0.1",
    "json-loader": "0.5.4",
    "style-loader": "0.13.1",
    "webpack": "1.13.1",
    "webpack-dev-server": "1.14.1",
    "react": "15.2.1",
    "react-dom": "15.2.1",
    "react-redux": "4.4.5",
    "react-router": "2.6.0",
    "redux": "3.5.2",
    "redux-actions": "0.10.1"
  }
}
```

定义脚本,使用concurrently工具来构建并运行Webpack 开发服务器

安装多种Babel插件、加载器和模块

安装concurrently来加快npm脚本的运行

安装extract-text-webpack插件,将内联样式合并到包中

安装react-redux来处理数据

安装redux-actions来更好地组织 Redux reducer



由于使用 Webpack 来打包依赖，因此所有必需的包都在 `bundle.js` 中。将所有的依赖都放在 `devDependencies` 中(我对部署的包很挑剔，不希望在部署环境中出现任何只存在却未使用的模块，造成安全漏洞)。当使用 `--production` 标记时，`npm` 忽略 `devDependencies`，就像使用 `npm i--production` 一样。

接下来通过创建 `webpack.config.js` 来定义构建过程，见代码清单 14.2(ch14/redux-netflix/webpack.config.js)

代码清单 14.2 用于Netflix克隆版的Webpack配置文件

```
const path = require('path')
const ExtractTextPlugin = require('extract-text-webpack-plugin')

module.exports = {
  entry: {
    index: [
      'babel-polyfill',
      './src/index.js'
    ]
  },
  output: {
    path: path.join(__dirname, 'build'),
    filename: '[name].js'
  },
  target: 'web',
  module: {
    loaders: [
      {
        loader: 'babel-loader',
        include: [path.resolve(__dirname, 'src')],
        exclude: /node_modules/,
        test: /\.js$/,
        query: {
          presets: ['react', 'es2015', 'stage-0']
        }
      }, {
        loader: 'json-loader',
        test: /\.json$/
      }, {
        loader: ExtractTextPlugin.extract('style',
          'css?modules&localIdentName=[local]__[hash:base64:5]'),
        test: /\.css$/,
        exclude: /node_modules/
      }
    ]
  },
  resolve: {
    modulesDirectories: [
      './node_modules',
      './src'
    ]
  },
  plugins: [
    new ExtractTextPlugin('styles.css')
  ]
}
```

指定入口(并不一定必须是\*.jsx)

应用polyfill来完全模拟ES2015环境

使用path.join()指定输出文件夹，使其跨平台可用(例如在Windows中)

作为数组使用加载器

使用JSON加载器，从JSON文件获取电影的模拟数据库

指定Babel预设(即如何处理代码)

在插件中应用加载器来提取样式并将它们合并到一个文件(而不是许多文件)中

提供一个插件，来提取文本

这些配置足够了。在下一节中，将开始使用 `Redux`。

### 14.3.3 启用 Redux

为使 Redux 能够在 React 应用中工作, 组件的层次结构需要将 Provider 组件置于顶层。Provider 组件来自 react-redux 软件包, 并且将数据从数据存储注入组件。就是这样: 使用 Provider 作为顶层组件意味着所有的子节点都可以访问数据存储。这很优雅。

为了使 Provider 组件工作, 需要将数据存储作为 store 属性提供给它。Store 是一个表示应用状态的对象。Redux(react-redux)带有 createStore() 函数, 它从 ch14/redux-netflix/src/modules/index.js 获取 reducer 并返回 Store 对象。

为了呈现 Provider 组件及其整个组件的子树, 可以使用 react-dom 的 render()。它接收第一个参数(<Provider>)并将其加载到作为第二个参数传入的元素(document.getElementById('app'))中。

结合所有这些, 现在应用的入口应该如代码清单 14.3 所示(ch14/redux-netflix/index.js)。可以通过 JSX 格式传递 Store 实例(使用 reducer)来定义 Provider 组件。

代码清单14.3 主应用的入口点

```
const React = require('react')
const { render } = require('react-dom')
const { Provider } = require('react-redux')
const { createStore } = require('react-redux')
const reducers = require('./modules')
const routes = require('./routes')

module.exports = render((
  <Provider store={createStore(reducers)}>
    {routes}
  </Provider>
), document.getElementById('app'))
```

为了使整个应用能够使用 Redux 功能, 需要在子组件中实现一些代码, 例如连接到数据存储。来自同一个 react-redux 包的 connect() 函数接收一些参数。它返回一个函数, 然后封装组件, 这样它就可以接收数据存储中的某些部分到它的属性中。一会儿你将看到这一点。

现在完成了 index.js。Provider 组件负责将数据从数据存储传递到所有连接的组件, 所以不需要直接传递属性。但是有一部分丢失了, 例如路由、reducer 和操作, 下面一个一个地检查它们。

### 14.3.4 路由

使用 react-router, 可以为每个浏览器地址声明一个组件层次结构。在第 13 章介绍了 React Router, 所以你应该熟悉它; 用它进行客户端路由。React 路由并不严格与服务器端路由对应, 但是你有时候可能想这样做, 特别是与第 16 章中讨论的技术相结合。

React Router 的要点是每个路由都可以由几个嵌套的 Route 组件声明, 每个组件有两个属性:

- path——可包含 URL 参数的 URL 路径或地址, 例如/movies/:id 格式的 localhost:8080/movies/1021。使用/可以定义与父路由路径无关的路径, 例如/:id 格式的 localhost:8080/1012。

- **component**——当用户跳转到路径/地址时，将要渲染的组件的引用。向上直到 **Provider** 的所有父组件也将被渲染。例如，在代码清单 14.4 中，转到 `localhost:8080/movies/1021` 将会渲染 **Movie**、**Movies** 和 **App** 组件。

需要在根路径和 `/movies` 路径上显示一组电影封面。另外，需要在 `/movies/:id` 路径上显示给定电影的信息。路由配置使用 **IndexRoute**，如代码清单 14.4 所示(ch14/redux-netflix/src/ routes.js)。

代码清单 14.4 使用 React Router 定义 URL 路由

```
const React = require('react')
const {
  Router,
  Route,
  IndexRoute,
  browserHistory
} = require('react-router')
const App = require('components/app/app.js')
const Movies = require('components/movies/movies.js')
const Movie = require('components/movie/movie.js')

module.exports = (
  <Router history={browserHistory}>
    <Route path="/" component={App}>
      <IndexRoute component={Movies} />
    <Route path="movies" component={Movies}>
      <Route path=":id" component={Movie} />
    </Route>
  </Route>
</Router>
)
```

定义索引路由：空 URL 的路由

为路由器提供浏览器记录或哈希记录

使用冒号语法：id 定义 URL 参数为电影的 id

**IndexRoute** 和 **Route** 都嵌套在最上面的路由中。这使得根路由和 `/movies` 路由都渲染 **Movies** 组件。单个电影的视图需要电影 id 来从 **Redux** 的数据存储获取有关该特定电影的信息，因此需要定义带 URL 参数的路径。为此，请使用冒号语法： `path=":id"`。图 14.9 展示了单个视图及其 URL 如何在小屏幕上显示，这要归功于响应式 **CSS**。注意，URL 是 `movies/8`，其中 8 是电影 id。接下来，你将看到如何使用 **Redux Reducer** 获取数据。

### 14.3.5 合并 reducer

考虑 `src/index.js` 中的 `createStore()` 函数所使用的模块：

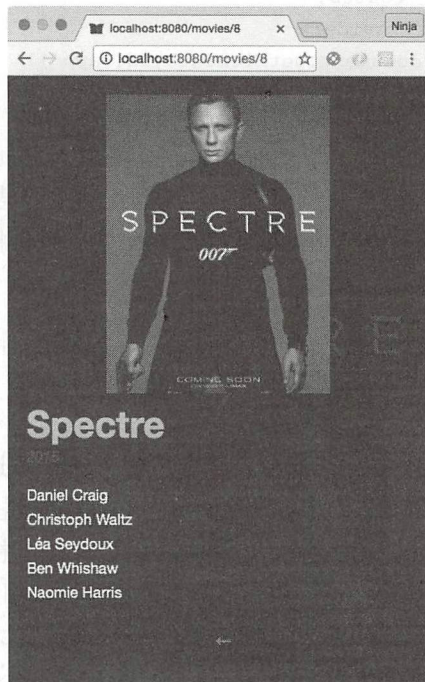


图 14.9 小屏幕上的单个电影视图，URL 中包含电影 id



```

...
const reducers = require('./modules')
...

module.exports = render((
  <Provider store={createStore(reducers)}>
    {routes}
  </Provider>
), document.getElementById('app'))

```

从./modules(./modules/index.js) 导入(合并)reducer

使用reducer

这有什么用处？需要在数据存储中存储电影数据。也许将来会实现数据存储的其他部分，例如用户账户或其他实体。因此，可以使用 Redux 的如下特性：允许根据需要按照意愿创建数据存储的不同部分，尽管此时只需要一个。从某种意义上来说，通过执行合并 reducer 的中间步骤，可以创建更好的架构。以便稍后可以通过向 ./modules/index.js(或./modules)添加更多的 reducer 来轻松扩展应用，使用插件 Node 模式<sup>4</sup>。这种方法也被称作分割 reducer (<http://mng.bz/Wprj>)

每个 reducer 都可以改变数据存储中的数据；但为了使操作安全，可能需要将应用状态分为不同的部分，然后将它们组合成一个数据分割。这种分而治之的方法被推荐用于大型应用，在这些应用中会有越来越多的 reducer 和操作。使用 Redux(ch14/redux-netflix/src/modules/index.js)中的 combineReducers()函数可以很容易将多个 reducer 合并，参见代码清单 14.5。

代码清单14.5 合并reducer

```

const { combineReducers } = require('redux')
const {
  reducer: movies
} = require('./movies')

module.exports = combineReducers({
  movies
  // more reducers go here
})

```

从redux的combineReducers 属性导入combineReducers

应用ES6/ES2015解构赋值，从 ./movies.js的reducer属性创建一个名为movies的reducer对象

导出合并了电影的reducer

可以按照自己的意愿通过 reducer 在数据存储中创建独立的分支。可以随意命名。在这种情况下，名为 movies 的 reducer 在被导入后传给 combineReducers()函数，把结果作为普通对象的名为 movies 的属性。

如此，可以声明数据存储中的一个独立部分，并称之为 movies。对于 ./movies 中的 reducer 负责的每个操作，都只会接触到这一部分，而接触不到其他部分。

### 14.3.6 电影的 reducer

接下来实现电影的 reducer。Redux 中的 reducer 是每次分发操作时都会调用的函数。

4 参见 Azat Mardan 的“Node Patterns: From Callbacks to Observer”，webapplog，<http://mng.bz/p9vd>

它在执行时有两个参数：

- 第一个参数是 `state`，表示 `reducer` 管理的数据引用，属于整体状态的一部分。
- 第二个参数是 `action`，表示刚刚所分发操作的一个对象。

换句话说，`reducer` 的输入是以前操作的结果：当前的状态(`state`)和操作(`action`)。`Reducer` 获取当前状态并应用操作。结果是一个新的状态。如果 `reducer` 是没有(它们应该没有)副作用的纯函数，那么可以获得 `Redux` 和 `React` 的所有优点，例如热重载和时间旅行。

### JavaScript 中的 reducer

术语 `reducer` 来自函数式编程。`JavaScript` 有一些函数式特性，所以才有 `Array.reduce()`。

简而言之，`reduce` 方法是一个归并列表中项的操作过程，所以输入有多个值，并且输出是单个值。`reducer` 处理的列表可以是数组，与 `JavaScript` 一样，也可以是其他数据结构(如列表)，与 `JavaScript` 不同。

例如，可以返回名称列表中名称出现的次数。名称列表是输入，出现的次数是输出。

要使用 `reducer`，可以调用一个方法并传递一个可以接收以下参数的函数：

- 累加值——传递给下一次迭代的内容以及最终输出的内容。
- 当前值——列表项。

每次遍历列表(或 `JS` 中的数组)中的项时，`reducer` 函数将获取累加值。在 `JavaScript` 中，方法是 `Array.reduce()`。例如，要获取名称频率，可以运行以下 `reducer` 代码，此处使用了三元运算符，如果当前值(`curr`)为 `'azat'`，那么累加值(`acc`)<sup>5</sup>加 1：

```
const users = ['azat', 'peter', 'wiliam', 'azat', 'azat']
console.log(users
  .reduce((acc, curr) => (
    (curr == 'azat') ? ++acc : acc
  ), 0)
)
```

在 `Redux reducer` 中，累加值是状态对象，当前值是当前操作。函数的结果是新的状态。

**提示：**避免将 `API` 调用放在 `reducer` 中。记住，`reducer` 应该没有副作用的纯函数。它们是状态机，不应该做异步操作，比如对 `API` 的 `HTTP` 调用。放置这些异步调用类型的最好位置是中间件(<http://redux.js.org/docs/advanced/Middleware.html>)或作为操作创建器(<http://mng.bz/S3II>，操作创建器是一个创建操作的函数)的 `dispatch()`。你在本章稍后将会看到组件中的 `dispatch()`。

典型的 `reducer` 是如下包含庞大 `switch/case` 语句的函数：

```
const FETCH_MOVIES = 'movies/FETCH_MOVIES'
const FETCH_MOVIE = 'movies/FETCH_MOVIE'

const initialState = {
  movies: [],
  movie: {}
}
```

5 有关 `Array.prototype.reduce()` 的详细文档，请参阅 Mozilla 开发者网站 <http://mng.bz/Z55j>



```

function reducer(state = initialState, action) {
  switch(action.type) {
    case FETCH_MOVIES:
      return {
        ...state,
        all: action.movies
      }
    case FETCH_MOVIE:
      return {
        ...state,
        current: action.movie
      }
  }
}

module.exports = {
  reducer
}

```

在数据存储中保存或更改所有电影的列表

ES6扩展运算符，按照键值传递状态对象

在数据存储中保存或修改某一电影

用ES6语法导出包含 reducer 方法的对象

但是按照业内权威 Douglas Crockford 在其经典著作 *JavaScript: The Good Parts* (O'Reilly Media, 2008) 中的观点，使用 `switch/case` 被认为是一种不好的实践。有一个方便的 `redux-actions` 库(<https://github.com/acdlite/redux-actions>)，可以将 `reducer` 函数变成一个更整洁、更实用的表单。可以使用更健壮的对象，替换庞大的 `switch/case` 语句。

下面使用 `redux-actions` 中的 `handleActions`。它有一个类似映射的普通对象，将操作类型作为键，将函数作为值。这样，每种操作只有一个函数会被调用。换句话说，这个函数就是根据操作类型选择的。

前面代码片段中的函数可以使用 `redux-actions` 和 `handleActions` 来重写，如代码清单 14.6 所示(ch14/redux-netflix/src/modules/movies.js):

代码清单 14.6 使用 `redux-actions` 库

```

const { handleActions } = require('redux-actions')

const FETCH_MOVIES = 'movies/FETCH_MOVIES'
const FETCH_MOVIE = 'movies/FETCH_MOVIE'

const initialState = {
  movies: [],
  movie: {}
}

module.exports = {
  fetchMoviesActionCreator: (movies) => ({
    type: FETCH_MOVIES,
    movies
  }),
  fetchMovieActionCreator: (index) => ({
    type: FETCH_MOVIE,
    index
  }),
  reducer: handleActions({
    [FETCH_MOVIES]: (state, action) => ({
      ...state,

```

定义 `FETCH_MOVIES` 操作创建器，它将返回一个操作对象

定义 `FETCH_MOVIE` 操作创建器，它将返回一个操作对象



```

    all: action.movies
  )),
  [FETCH_MOVIE]: (state, action) => ({
    ...state,
    current: state.all[action.index - 1]
  })
}, initialState)
}

```

在 Movies 组件中  
获取所有的电影

在 Movie 组件中通过  
index(作为电影id的  
URL参数)获取当前  
的电影

这段代码看起来类似 `switch/case`，但更多的是将函数映射到操作，而不是在可能十分庞大的条件语句中选择它们。

### 14.3.7 操作

要更改数据存储中的数据，需要使用操作。在此阐明，操作可以是任何东西，而不仅是浏览器中的用户输入。例如，可能是异步操作的结果。基本上，任何代码都可以成为操作。操作是数据存储的唯一信息来源，这些数据被从应用发送到数据存储。操作通过前面提到的 `store.dispatch()` 来执行，或者通过 `connect()` 来执行。在介绍如何调用操作之前，先讨论操作的类型。

每个操作都由一个普通的对象表示，至少有一个属性 `type`。操作可以具有任意数量的其他属性，通常用来将数据传递到数据存储。所以，每个操作都有类型，像下面这样：

```

{
  type: 'movies/I_AM_A_VALID_ACTION'
}

```

此处，操作的类型是字符串。

**注意：**一般使用模块名小写在前、大写的操作类型在后的方式命名操作。如果能保证不发生冲突，也可以省略模块名。

在现代 **Redux** 开发中，操作类型被声明为常量字符串：

```

const FETCH_MOVIES = 'movies/FETCH_MOVIES'
const FETCH_MOVIE = 'movies/FETCH_MOVIE'

```

这里声明了两种操作。两者都是由两部分组成的字符串：**Redux** 模块的名称和操作类型名称。当不同 **reducer** 中的操作有相似的命名时，这种做法很有用。

每次想改变应用的状态时，都需要分发相应的操作。相应的 **reducer** 会执行，最终得到应用更新后的状态。考虑从 **API** 或用户表单中收到的数据：所有数据都可以在数据存储中存储和更新。例如：

```

this.props.dispatch({
  type: FETCH_MOVIE,
  movie: {}
})

```

一系列步骤如下：

- ① 在一个组件中，使用一个操作对象作为参数调用 `dispatch()`，如果需要，该操作对象可以有 `type` 属性和数据。
  - ② 在 `reducer` 模块中执行相应的 `reducer`。
  - ③ 在数据存储中更新状态，此状态可以在组件中使用。
- 稍后再进行分发，下面来看看如何在组件中通过使用操作类型来避免传递。

### 14.3.8 操作创建器

要更改数据存储中的任何内容，需要执行一个通过所有 `reducer` 的操作。然后 `reducer` 根据操作类型修改应用的状态。出于这个原因，必须知道操作类型。但捷径是可以将操作的类型隐藏在操作创建器下。总的来说，步骤如下：

- ① 使用数据(如果需要的话)调用操作创建器，操作创建器可以在 `reducer` 模块中定义。
- ② 在组件中分发一个操作，不需要操作类型。
- ③ 在 `reducer` 模块中执行相应的 `reducer`。
- ④ 在数据存储中更新状态。

如下所示：

```
this.props.dispatch(fetchMoviesActionCreator({movie: {}}))
```

简而言之，操作创建器是一个函数，该函数返回一个操作，如下所示：

```
function fetchMoviesActionCreator(movies) {  
  return {  
    type: FETCH_MOVIES,  
    movies  
  }  
}
```

通过操作创建器，可以将复杂的逻辑隐藏在函数调用中。在这种情况下，不需要逻辑。该函数执行的唯一操作是返回一个操作：一个普通对象，带有定义了此操作的 `type` 属性，以及带有包含所有电影信息的值类型为数组的 `movies` 属性。如果需要扩展 Netflix 的克隆版以便可以添加电影，那么需要添加名为 `addMovie()` 的操作创建器：

```
function addMovie(movie) {  
  return {  
    type: ADD_MOVIE,  
    movie  
  }  
}
```

或者使用 `watchMovie()`：

```
function watchMovie(movie, watchMovieIndex, rating) {  
  return {  
    type: WATCH_MOVIE,  
    movie,  
    watchMovieIndex,  
    rating  
  }  
}
```

```
    index: watchMovieIndex,  
    rating: rating,  
    receivedAt: Date.now()  
  }  
}
```

记住，操作必须有 `type` 属性！

为了能分发操作，必须将组件连接到数据存储。这将更有趣，因为已经接近状态更新。

### 14.3.9 将组件连接到数据存储

现在你已经学会了如何将数据放入数据存储，下面介绍如何从组件访问存储的数据。幸运的是，`Provider` 组件有将数据带入组件的功能。但要访问数据，需要明确地将组件连接到数据存储。

默认情况下，组件并未连接到数据存储；并且将数据存储放在顶级 `Provider` 组件的层次结构中是不够的。为什么？考虑明确地连接某些组件。

是否还记得，根据 `React` 最佳实践，有两种类型的组件：展示(木偶型)组件和容器(智能)组件，参见第 8 章。展示组件不需要存储，它们应该只应用属性。同时，容器组件需要存储和分发器。甚至在 `Redux` 文档中，容器组件的定义也指出它们是数据存储的订阅者(<http://mng.bz/p4f9>)。`Provider` 组件所做的全部事情就是自动为所有的组件提供数据存储，以便其中的一些组件能够订阅/连接到数据存储。因此，对于容器组件，需要 `Provider` 组件和数据存储。

总而言之，已连接的组件可以在其属性中访问数据存储中的任何数据。如何将组件连接到数据存储？当然用 `connect()` 方法！感到困惑吗？来看一个例子。考虑根组件 `App`，它将使用 `Movies` 组件，至少应该用下面这段代码来显示电影列表(实际的 `Movies` 组件有更多的代码)：

```
class Movies extends React.Component {  
  render() {  
    const {  
      movies = []  
    } = this.props  
    return (  
      <div className={styles.movies}>  
        {movies.map((movie, index) => (  
          <div key={index}>  
            {movie.title}  
          </div>  
        ))}  
      </div>  
    )  
  }  
}
```

目前，虽然以 `Provider` 作为父组件，但 `Movies` 组件并未连接到数据存储。可通过下面的代码片段来连接。`connect()` 函数由 `react-redux` 包提供并且最多可以接收 4 个参数，但目前你只能使用一个参数：



```
const { connect } = require('react-redux')
class Movies extends React.Component {
  ...
}
module.exports = connect()(Movies)
```

`connect()` 函数返回一个应用于 `Movies` 组件的函数。结果不是导出 `Movies`，而是调用 `connect()` 来处理 `Movies`，并以 `Provider` 作为父组件，所以 `Movies` 组件连接到了数据存储。

现在 `Movies` 组件可以从数据存储接收任何数据并分发操作(没看到这条消息，是吗？)。但是，要以需要的格式接收数据，必须通过创建一个简单的映射函数(表达式是更精确的术语，因为需要返回结果)将状态映射到组件属性。

在一些教程中，你可能会看到一个名为 `mapStateToProps()` 的函数，但它不一定是显式声明的函数。使用匿名箭头函数同样简洁明了。这个映射函数被传入来自最喜欢的 `react-redux` 的特殊方法 `connect()`。请记住，`state`(状态)是 `connect()` 的第一个参数：

```
module.exports = connect(function(state) {
  return state
})(Movies)
```

或使用下面这种优秀、流行、下一代 ES 式的并且对 `React` 友好的隐式返回模式：

```
module.exports = connect(state => state)(Movies)
```

使用该设置，可以从数据存储中获取整个应用的状态并将之放入 `Movies` 组件的属性中。你会发现，通常只需要状态的一个有限子集。在本例中，`Movies` 只需要 `movies.all`：

```
class Movies extends React.Component {
  render() {
    const {
      children,
      movies = [],
      params = {}
    } = this.props
    ...
  }

  module.exports = connect(({movies}) => ({
    movies: movies.all
  })), {
    fetchMoviesActionCreator
  })(Movies)
```

下面是 `Movie` 组件中的代码片段，仅从状态映射 `movies.current`：

```
class Movie extends React.Component {
  render() {
    const {
      movie = {
        starring: []
      }
    } = this.props
    ...
  }
}
```

```
module.exports = connect(({movies}) => ({
  movie: movies.current
}), {
  fetchMovieActionCreator
})(Movie)
```

你会看到，如果数据存储是空的，组件将不会收到任何额外的属性，因为没有任何东西。

接下来发生的是 **Redux** 的一些神奇的事情：每次数据存储的一部分被更新时，依赖该部分的所有组件都接收到新的属性，因此被重新渲染。这种情况发生在分发每一个操作时，这意味着：现在，组件松散地相互依赖，并且只有在数据存储更新时才更新。任何组件都可以通过分发适当的操作来引起这样的更新。没必要使用传统的回调函数作为属性，并将它们从最顶层的组件传到最深嵌套的组件；只需要连接组件到数据存储。

### 14.3.10 分发操作

要修改数据存储中的数据，需要分发操作。一旦将组件连接到数据存储，就可以收到已经映射到某些应用状态的属性，而且还会收到 `dispatch` 属性。

`dispatch()` 方法是一个函数，它将操作作为参数并分发(发送)操作到数据存储。因此，可以通过执行 `this.props.dispatch()` 并使用操作作为参数来分发操作：

```
componentWillMount() {
  this.props.dispatch({
    type: FETCH_MOVIE,
    movie: {}
  })
}
```

`type` 是一个字符串，**Redux** 库将其应用于匹配此类型的所有 **reducer**。分发操作后，意味着已经更改了数据存储，所有连接到数据存储的组件，以及从应用的数据存储的某部分映射了属性的组件，都将重新渲染。没必要检查组件是否需要更新或再做任何事情。可以依赖 `render()` 函数中的新属性：

```
class Movie extends React.Component {
  render() {
    const {
      movie = {
        starring: []
      }
    } = this.props
    ...
```

可以使用操作创建器(`fetchMovieActionCreator()`函数)替换纯操作(带有 `type` 属性的对象)：

```
const fetchMovieActionCreator = (response) => {
  type: FETCH_MOVIE,
  movie: response.data.data.movie
```

```

}
...
componentWillMount() {
  ... // Make AJAX/XHR request
  this.props.dispatch(fetchMovieActionCreator(response))
}

```

因为 `fetchMovieActionCreator()` 返回一个普通对象，该对象与上例中的对象(键值为 `type` 和 `movie`)相同，所以可以调用此操作创建器——`fetchMovieActionCreator()` 函数，并将结果传给 `dispatch()`：

- ① 异步获取数据(响应)。
- ② 创建操作(`fetchMovieActionCreator()`)。
- ③ 分发操作(`this.props.dispatch()`)。
- ④ 执行 `reducer`。
- ⑤ 在属性(`this.props.movie`)中更新状态。

#### 14.3.11 将操作创建器传递到组件属性中

可以在组件文件中定义操作创建器作为函数，但是还有另一种使用操作创建器的方式：在模块中定义它们，导入它们并将它们放入组件属性中。为此，可以使用 `connect()` 函数的第二个参数并将操作创建器作为方法传递：

```

const {
  fetchMoviesActionCreator
} = require('modules/movies.js')
class Movies extends Component {
  ...
}
module.exports = connect(state => ({
  movies: state.movies.all
}), {
  fetchMoviesActionCreator
})(Movies)

```

从client/modules/movies.js  
导入操作创建器

将数据映射到movies属性

现在可以通过属性来引用 `fetchMovieActionCreator()` 并且不必使用 `dispatch()` 来传递操作，像下面这样：

```

class Movies extends Component {
  componentWillMount() {
    this.props.fetchMoviesActionCreator()
  }
  render() {
    const {
      movies = []
    } = this.props
    return (
      <div className={styles.movies}>
        {movies.map((movie, index) => (
          <div key={index}>

```

直接调用操作创建器  
来派发操作

将movies赋值为this.props.movies  
或空数组(ES6解构赋值)



```

        {movie.title}
      </div>
    )}
  </div>
)
}
}

```

这个新的操作创建器被自动封装在一个有效的 `dispatch()` 调用中。不需要担心，可自行完成。你现在应用清楚了位于 `ch14/redux-netflix/src/components/movies/movies.js` 的 `Movies` 组件是如何实现的。

为了清晰起见，可以重命名 `fetchMoviesActionCreator()` 为 `fetchMovies()` 或者像下面这样做：

```

const {
  fetchMoviesActionCreator
} = require('modules/movies.js')
class Movies extends Component {
  componentWillMount() {
    this.props.fetchMovies()
  }
  ...
  module.exports = connect(state => ({
    movies: state.movies.all
  }), {
    fetchMovies: fetchMoviesActionCreator
  })(Movies)
}

```

使用 `fetchMovies()` 派发

重命名操作方法

`connect()` 的第一个参数是将状态映射到组件属性的函数，它将整个状态(`state`)作为唯一的参数，并返回一个具有单个属性的普通对象 `movies`：

```

...
module.exports = connect(state => ({
  movies: state.movies.all
}), {
  fetchMoviesActionCreator
})(Movies)

```

对 `state.movies` 使用解构赋值，可以使代码更具表现力：

```

module.exports = connect(({movies}) => ({
  movies: movies.all
}), {
  fetchMoviesActionCreator
})(Movies)

```

在 `Movies` 组件的 `render()` 函数中，`movies` 的值是从属性获得的，并且被渲染到一组兄弟关系的 DOM 元素中。它们中的每个都是一个 `div` 元素，其中的文本被设置为 `movie.title` 的值。这是一种典型的渲染数组到 DOM 元素片段中的方法。

想知道最终的 `Movies` 组件是什么样子吗？详见代码清单 14.7(`ch14/redux-netflix/src/components/movies/movies.js`)。

## 代码清单14.7 传递操作创建器到Movies属性中

```

const React = require('react')
const { connect } = require('react-redux')
const { Link } = require('react-router')
const movies = require('../../movies.json')
const {
  fetchMoviesActionCreator
} = require('modules/movies.js')
const styles = require('./../movies.css')

class Movies extends React.Component {
  componentWillMount() {
    this.props.fetchMovies(movies)
  }

  render() {
    const {
      children,
      movies = [],
      params = {}
    } = this.props

    return (
      <div className={styles.movies}>
        <div className={params.id ? styles.listHidden : styles.list}>
          {movies.map((movie, index) => (
            <Link
              key={index}
              to={`../movies/${index + 1}`}>
                <div
                  className={styles.movie}
                  style={{ backgroundImage: `url(${movie.cover})` }} />
              </Link>
            ))}
        </div>
        {children}
      </div>
    )
  }
}

module.exports = connect(({movies}) => ({
  movies: movies.all
}), {
  fetchMovies: fetchMoviesActionCreator
})(Movies)

```

从JSON文件下载模拟数据库(感谢json-loader)到movies中

使用fetchMoviesActionCreator()派发一个操作(FETCH\_MOVIES),此操作带有从JSON对象movies获取的数据,此JSON对象用于取代向API服务器发送AJAX/XHR请求

按照React Router层次结构中的定义传递子组件

将组件连接到数据存储,这样可以在属性中提供对数据的访问以及名为fetchMoviesActionCreator()的操作创建器

由此可见,异步数据交换非常简单:执行异步调用(使用 fetch() API、axios 等),然后在 componentWillMount() 中分发操作。

在 React 团队推荐的 componentDidMount() 中调用 AJAX/XHR 会更好:

```

componentWillMount() {
  // this.props.fetchMovies(movies)
}

componentDidMount() {

```

没有分发数据,使用请示(异步)导入数据

```

    fetch('/src/movies.json', {method: 'GET'})
      .then((response)=>{return response.json()})
      .then((movies)=>{
        this.props.fetchMovies(movies)
      })
  }

```

取得由Webpack 开发服务器(异步)提供的JSON文件

分发的操作携带通过异步GET请求从服务器获取的数据

可以像通过 GET 那样通过 POST、PUT 或其他 HTTP 调用来做同样的事情。在下一章中将会提到这些调用。

我们已经完成了 Movies 组件。接下来，将介绍 Movie 组件——但只是简单地介绍一下，因为大部分的 Redux 连接与 Movies 中的相似。所不同的是，Movie 将会获取 URL 参数中的 movie id。React Router 将它放在 `this.props.params.id` 中。这个 id 将通过操作分发，并在 reducer 中用来过滤出单个电影。在此提醒，这些是来自 `src/modules/movies.js` 的 reducer:

```

...
reducer: handleActions({
  [FETCH_MOVIES]: (state, action) => ({
    ...state,
    all: action.movies
  }),
  [FETCH_MOVIE]: (state, action) => ({
    ...state,
    current: state.all[action.index - 1]
  })
},
),
...

```

使用movie索引来返回单个电影

现在来看 Movie 的实现，它从 React Router 的 URL 参数获取 movie id，并将此 id 作为索引来使用，得到一个与前面不同的从状态到属性的映射，如代码清单 14.8 所示 (`src/components/movie/ movie.js`)。

代码清单14.8 Movie的实现

```

const React = require('react')
const { connect } = require('react-redux')
const { Link } = require('react-router')
const {
  fetchMovieActionCreator
} = require('modules/movies.js')
const styles = require('./movie.css')

class Movie extends React.Component {
  componentWillMount() {
    this.props.fetchMovie(this.props.params.id)
  }
  componentWillUpdate(next) {
    if (this.props.params.id !== next.params.id) {
      this.props.fetchMovie(next.params.id)
    }
  }
  render() {

```

导入CSS文件

只有当URL参数改变时才分发操作



```

const {
  movie = {
    starring: []
  }
} = this.props

return (
  <div
    className={styles.movie}
    style={{backgroundImage: `linear-gradient(90deg, rgba(0, 0, 0, 1) 0%,
    rgba(0, 0, 0, 0.625) 100%), url(${movie.cover})`}}>
    <div
      className={styles.cover}
      style={{backgroundImage: `url(${movie.cover})`}} />
    <div className={styles.description}>
      <div className={styles.title}>{movie.title}</div>
      <div className={styles.year}>{movie.year}</div>
      <div className={styles.starring}>
        {movie.starring.map((actor = {}, index) => (
          <div
            key={index}
            className={styles.actor}>
              {actor.name}
            </div>
          )))
      </div>
    </div>
    <Link
      className={styles.closeButton}
      to="/movies">
        <←
      </Link>
    </div>
  )
}

module.exports = connect(({movies}) => ({
  movie: movies.current
}), {
  fetchMovie: fetchMovieActionCreator
})(Movie)

```

对元素应用内联样式

将数据从reducer映射到属性

### 14.3.12 运行 Netflix 的克隆版

是时候运行这个项目了。当然，也可以从一开始就这样做，因为启动脚本在 `package.json` 中。此脚本使用 `npm` 库来同时运行两个进程：以 `watch` 模式运行 `Webpack` 构建和 `Webpack` 开发服务器（端口 8080）：

```

"start": "concurrently \"webpack --watch --config webpack.config.js\"
  & \"webpack-dev-server\""

```

导航到项目根目录(`ch14/redux-netflix`)。使用 `npm i` 安装依赖关系，并在项目文件夹下运行此项目：`npm start`。使用你最喜欢的浏览器打开 `http://localhost:8080`。

单击查看路由是否正常工作、图片是否加载，而无论是使用模拟数据(`require()`)还是通

过 GET 请求加载。注意，如果当前页面位于 `http://localhost:8080/movies/1` 并且刷新页面，将看不到任何内容。可以在下一章中注意这一点，将会实施 Node 和 Express 服务器来支持非哈希 URL。

### 14.3.13 Redux 总结

Redux 提供了一个位置来存储整个应用的数据；改变数据的唯一方法是通过操作。这使得 Redux 具有通用性——可以在任何地方使用，而不仅仅是在 React 应用中。但是通过 `react-redux` 库，可以使用 `connect()` 函数将任何组件连接到数据存储并对其中的任何更改做出反应。

这是响应式编程的基本思想：实体 A 观察实体 B 中的变化，当变化发生时对这些变化做出响应，但反过来不成立。这里，实体 A 可以是任何组件，实体 B 是数据存储。

在连接(`connect()`)组件并将数据存储的属性映射到组件的属性(`this.props`)时，可以在 `render()` 函数中引用后者。通常，需要先更新数据存储中的数据，才能引用数据。这就是在组件的 `componentWillMount()` 函数中调用操作的原因。当组件第一次加载并渲染时，组件所引用的数据存储中的部分可能为空。但是一旦数据存储中的数据被更新，它就会被保留。这就是在 Netflix 克隆版中，在导航应用的位置(页面或视图)之后，电影列表仍然保持不变的原因。是的，组件卸载后数据不会从数据存储中消失，这与使用组件的状态(还记得 `this.state()` 和 `this.setState()` 吗?)时不同。这样，Redux 数据存储可以为应用程序中需要相同数据的不同部分提供服务，而无须重新加载数据。

在 `render()` 函数中通过分发操作来更新组件属性也是安全的，因为这项操作是延迟执行的。另一方面，在没有 Redux 的情况下，组件可能更新的任何时刻——`render()`、`componentWillMount()` 或 `componentWillUpdate()`，都不能使用 `setState()`。Redux 的这个特性增强了它的灵活性。

## 14.4 测验

1. 命名 JavaScript 中 `Array.reduce()` 方法的归并函数(回调)的两个主要参数。
2. Redux 相比 Facebook 的 Flux(flux)提供了简单且更大的生态系统以及更好的开发体验，这么理解正确还是错误？
3. 下面的哪个可以被用来创建数据存储和 provider？`new Provider(createStore(reducers))`、`<Provider store={createStore(reducers)}>` 还是 `provider(createStore(reducers))`？
4. Redux 需要分发器，因为那是 Flux 定义的，这么理解正确还是错误？
5. 在本项目中，`movies.all` 获取所有的电影，且 `movies.current` 获取当前的电影。在 `connect()` 调用中，它们分别在 `Movies` 和 `Movie` 组件中使用。在哪里定义 `movies.all` 和 `movies.current` 的逻辑？

## 14.5 小结

- 单向数据流为 React 应用提供可预测性和易维护性。
- 在使用 React 和单向数据流时，推荐 Flux 架构。
- Redux 是 Flux 架构最流行的实现之一。
- 使用 Redux，可以分发操作或将其放在属性对象中。
- Redux 的 `connect()` 提供了访问数据存储中数据和分发操作的功能——容器(智能)组件必要的功能。
- Redux Provider 组件提供子节点对数据存储中数据的访问，因此不必在属性中手动传递数据存储。
- reducer 是具有归并函数的文件，归并函数(通常)使用 `switch/case` 语句或 `handleActions` 将操作应用到新的状态：输入当前状态和操作，输出新的状态。
- Redux 的 `combineReducers` 能方便地合并多个 reducer，可以将这些 reducer 的代码分割成不同的模块/文件。

## 14.6 测验答案

1. 累加器的值和当前值是两个主要参数，没有它们就不能汇总列表。
2. 正确。见本章介绍和 Stack Overflow 上由 Dan Abramov 发布的 *Why Use Redux over Facebook Flux?*(<http://mng.bz/z9ok>)
3. `<Provider store={createStore(reducers)}>`
4. 错误。Redux 遵循 Flux，但不需要分发器，因此 Redux 更易于实现。
5. 在 `src/modules/movies.js` 的 reducer 中。



# 第 15 章

## 使用 GraphQL 处理数据

### 本章内容：

- 使用 GraphQL 和 axios 请求服务器端的数据
- 使用 Redux 数据存储管理数据
- 使用 Node/Express 实现 GraphQL 后端
- 支持无哈希的 URL 路由

在第 14 章中，使用 Redux 实现了一个仿照 Netflix 网站的应用。它的数据来源于我们本地的静态 JSON 文件，虽然可以通过使用 axios 或 fetch() 的 RESTful API 来提供数据，但是本章我们选择使用更加流行的 GraphQL 来向前端应用提供数据。

到目前为止，你一直在导入 JSON 文件作为后端数据仓库，或者模拟 GET RESTful 接口进行 RESTful 调用来获取相同的文件。我们称这种方式为模拟 API。模拟 API 有利于原型设计，因为已经准备好了前端工作，当需要持久存储时，可以直接使用后端服务器替换模拟服务，后端服务通常是 REST API(或者不得不使用 SOAP<sup>1</sup>)。

想象一下 Netflix 克隆版应用的 API 由另一支团队开发。在几次会议上你同意使用 JSON(或 XML)格式的数据。数据由他们提供。接口调通了，前端应用也得到了所有的数据。然后产品负责人和客户交谈，并决定添加客户想要的用来展示电影明星和评级的新字段。当需要额外的字段时会发生什么？这时不得不实现一个新的接口，如 `movies/:id/ratings`，或者由后端团队在旧的接口上添加新的字段。

也许应用仍然处在原型设计阶段。如果是这样，该字段可能会被添加到现有的 `movies/:id` 接口中。很容易看到随着时间的推移，会有更多的更改格式和结构的需求。但是电影评级必须出现在 `movies` 中，怎么办？或者，如果需要来自其他集合的新的嵌套字段(例

---

1 SOAP 是过时的协议，它严重依赖于 XML，现在已被 REST 替代

如朋友推荐), 又怎么办? 在快速敏捷开发和精益创业的时代, 灵活性是一项优势。接口字段和数据越快地适应终端产品越好。使用 GraphQL 的优雅解决方案可以扫清许多这些障碍。

## 15.1 GraphQL

在本章中, 将添加服务器来继续开发 Netflix 克隆版应用。该服务器将提供一个 GraphQL API, 它是将数据暴露给 React 应用程序的现代方式。GraphQL 经常与 Relay 一起使用, 但是正如你所看到的, 在本例中将使用 Redux 或其他浏览器数据管理库, 并使用 axios 进行 AJAX/XHR/HTTP 请求。

当使用 GraphQL 和 Redux 时, 不一定要使用 Node.js, 可以使用任何语言(Ruby、Python、Java、Go、Perl)来构建服务器(后端和 Web 服务器), 但是我个人还是推荐使用 Node, 在本节中我们使用的也是 Node, 因为它允许我们在整个开发技术栈中使用 JavaScript。

简而言之, GraphQL(<https://github.com/graphql/graphql-js>)使用由服务器(通常是 Node)解析的查询字符串, 而服务器又以这些查询指定的格式返回数据。查询是以类似 JSON 的格式编写的:

```
{
  user(id: 734903) {
    id,
    name,
    isViewerFriend,
    profilePicture(size: 50) {
      uri,
      width,
      height
    }
  }
}
```

而且响应也是传统但经典的 JSON 格式:

```
{
  "user" : {
    "id": 734903,
    "name": "Michael Jackson",
    "isViewerFriend": true,
    "profilePicture": {
      "uri": "https://twitter.com/mjackson",
      "width": 50,
      "height": 50
    }
  }
}
```

Netflix 克隆版应用的服务器可以使用 REST 或旧的 SOAP 标准。但是对于更新的

GraphQL 模式，可以通过让客户端(前端或移动应用)指定所需的数据而不是将逻辑编码到服务器端点/路由中进行逆向控制。这种倒置方法的一些优点如下<sup>2</sup>：

- 定制化查询：客户端获取完全符合他们需要的数据。
- 结构化的后端可使用任意代码：统一的 API 使服务器端更加灵活。
- 强类型：响应中更强大的数据验证和确定性，以及通过使用强类型语言(如 TypeScript、Swift、Java 和 Objective-C)使数据消耗更加简单。
- 层次化查询：查询返回的数据都是层次化的，这一点非常重要，因为数据是由层次化视图使用的。
- 更快的原型开发：因为只有一个查询端点，所以无需广泛的后端开发或大型独立的后端和 API 团队。
- 更少的 API 调用：前端应用减少了服务器请求，因为数据结构由前端应用指定，并且可以包含以前只能通过多个 REST 端点获取的内容。

### Relay 和 Relay Modern

可以在使用 Relay(<https://facebook.github.io/relay>, npm 上有 `graphql-relay-js` 和 `react-relay`) 的 React 应用中使用 GraphQL API。一些开发人员更喜欢在使用 GraphQL 时使用 Relay 而不是 Redux。如果查看 Relay 文档中提供的示例，可能会看到与 Redux 连接组件和替换存储的相似之处，以及使用远程 GraphQL API 服务。

React 允许通过组合许多简单的组件来构建复杂的 UI 和应用，Relay 允许组件指定所需的数据，以便数据需求变得本地化。React 组件不关心其他组件的逻辑和渲染。

Relay 的情况也是如此：组件保持它们的数据更接近自身，这样可以更轻松地组合组件(从简单的组件构建复杂的 UI 和应用)。

Relay Modern 是 Relay 的最新版本。它用起来更简单，扩展性也更强<sup>3</sup>。如果你或你的团队计划使用 GraphQL，那么强烈推荐使用 Relay/Relay Modern。

## 15.2 给 Netflix 克隆版应用添加服务器

使用由 Express(<https://github.com/expressjs/express>)和 GraphQL 开发的简单服务器，将数据传递给 React 应用。Express 在组织和提供 API 端点方面做得非常出色，而 GraphQL 负责使数据以一种浏览器比较友好的方式进行访问，如 JSON。

项目结构如下(可以重用 `redux-netflix` 中的很多代码)：

```
/redux-graphql-netflix
  /build
    /public
      index.js
      style.css
      server.js
```

编译后的文件  
编译后的前端文件  
编译后的后端文件

2 关于 GraphQL 的更多优势，请参见 Nick Schrock, “GraphQL 简介”, *React*, 2015 年 5 月 1 日, <http://mng.bz/DS65>

3 查看 <https://facebook.github.io/relay/docs/relay-modern.html>



```

/client
  /components
    /app
      app.css
      app.js
    /movie
      movie.css
      movie.js
    /movies
      movies.css
      movies.js
  /modules
    index.js
    movies.js
  index.js
  routes.js
/node_modules
/server
  index.js
  movies.json
  schema.js
index.html
package.json
webpack.config.js
webpack.server.config.js

```

前端React源代码文件

后端Express源代码文件

GraphQL数据模式

数据仍然从 JSON 文件中获取,但这次是服务器文件。可以轻松地使用 `server/schema.js` 中的数据库调用替换 JSON 文件 `movies.json`。但是在讨论 `schema` 之前,需要安装所有的依赖,包括 `Express`。

代码清单 15.1 显示了 `package.json` 文件(`ch15/redux-graphql-netflix/package.json`)。知道怎么做了吗?当然是复制文件并运行 `npm i` 进行安装。

代码清单15.1 Netflix克隆版应用的package.json文件

```

{
  "name": "redux-graphql-netflix",
  "version": "1.0.0",
  "description": "A sample project in React, GraphQL, Express and Redux that
    ➤ copies Netflix's features and workflow",
  "main": "index.js",
  "scripts": {
    "start": "concurrently \"webpack --watch --config webpack.config.js\"
      ➤ \"webpack --watch --config
      ➤ webpack.server.config.js\" \"webpack-dev-server\" \"nodemon
      ➤ ./build/server.js\"",
  },
  "repository": {
    "type": "git",
    "url": "git+https://github.com/azat-co/react-quickly.git"
  },
  "author": "Azat Mardan (http://azat.co)",
  "license": "MIT",
  "bugs": {
    "url": "https://github.com/azat-co/react-quickly/issues"
  },
  "homepage": "https://github.com/azat-co/react-quickly#readme",
  "devDependencies": {

```

添加start脚本,用来编译前后端代码并启动服务器

```

"babel-core": "6.11.4",
"babel-eslint": "6.1.2",
"babel-loader": "6.2.4",
"babel-polyfill": "6.9.1",
"babel-preset-es2015": "6.9.0",
"babel-preset-react": "6.11.1",
"babel-preset-stage-0": "6.5.0",
"concurrently": "2.2.0",
"css-loader": "0.23.1",
"eslint": "3.1.1",
"eslint-plugin-babel": "3.3.0",
"eslint-plugin-react": "5.2.2",
"extract-text-webpack-plugin": "1.0.1",
"json-loader": "0.5.4",
"nodemon": "1.10.0",
"style-loader": "0.13.1",
"webpack": "1.13.1",
"webpack-dev-server": "1.14.1",
"axios": "0.13.1",
"clean-tagged-string": "0.0.1-b6",
"react": "15.2.1",
"react-dom": "15.2.1",
"react-redux": "4.4.5",
"react-router": "2.6.0",
"redux": "3.5.2",
"redux-actions": "0.10.1"
},
"dependencies": {
  "express": "4.14.0",
  "express-graphql": "0.5.3",
  "graphql": "0.6.2"
}

```

添加nodemon开发工具，用来启动和重启Express服务(当服务器端代码发生更改时)

添加axios，用于前端发起HTTP请求(基于promise，类似于fetch)

添加程序，用于删除ES6字符串模板中的空格并执行其他清理操作

添加在后端使用的Express Node Web服务器框架

添加在前后端使用的Express的GraphQL插件

添加在前后端使用的GraphQL

接下来实现服务器文件 `server/index.js`。

### 15.2.1 在服务器端安装 GraphQL

由 Express 和 Node 实现的 Web 服务器的引擎是其起始点(有时被称为入口点): `index.js`。此文件位于服务器文件夹中，因为它仅在后端使用，并且因为安全性问题(它可以包含 API 密钥和密码)不能暴露给客户端。文件的顶层结构如下：

```

const path = require('path')
const express = require('express')
const graphqlHTTP = require('express-graphql')
// ...
const app = express()

app.use('/q',
  // ...
)

app.use('/dist',
  // ...
)

app.use('*',

```

导入依赖，包括 `express-graphql`

定义可以为各种数据提供服务的单一GraphQL路由

定义静态资源路由/dist

为不适用于/dist/\* URL的所有请求提供主HTML页面

```
// ...
})

app.listen(PORT, () => console.log(`Running server on port ${PORT}`))
```

└─ 启动服务器

让我们填一下缺失的这些部分。首先，请记住，对除了 API 端点和 bundle 文件之外的所有路由都需要返回相同的文件 `index.html`。这是必要的，因为当使用 HTML5 History API 并使用 `/movies/8` 这样的无哈希 URL 访问时，刷新页面可以使浏览器查询准确的位置。

你可能已经注意到，在以前的 Netflix 克隆版应用中，当刷新或重新加载单个电影页面时(例如 `/movies/8`)，页面没有显示任何内容。原因是需要为浏览器记录实现一些额外的代码。这些代码必须在服务器上，并且负责在所有请求(即使是 `/movies/8/`)上发送主 `index.html` 文件。

在 Express 中，当需要为每个路由声明单个操作时，可以使用 `*`(星号)：

```
app.use('*', (req, res) => {
  res.sendFile('index.html', {
    root: PWD
  })
})
```

将 HTML 文件发送到任何位置(`*`匹配)并不奏效，最终会出现 404 错误，因为这个 HTML 包含对编译的 CSS 和 JS 文件的引用(`/dist/styles.css` 和 `/dist/index.js`)。所以，需要先捕获这些静态文件请求：

```
app.use('/dist/:file', (req, res) => {
  res.sendFile(req.params.file, {
    root: path.resolve(PWD, 'build', 'public')
  })
})
```

另外，建议使用名为 `express.static()` 的 Express 中间件，如下所示：

```
app.use('/dist',
  express.static(path.resolve(PWD, 'build', 'public'))
)
```

提示：更多有关 Express 的中间件和技巧的信息，请参阅附录 C 和我的著作 *Pro Express.js* (Apress,2014)以及 *Express Deep API Reference*(Apress,2014)。

### static、public 和 dist 文件夹

建立公共文件夹的重要性不是被夸大。如果不将服务资源(例如文件)的行为限制到子文件夹(比如 `dist` 或 `public`)，那么所有代码都将暴露给任何访问服务的人员。如果放弃使用子文件夹，那么后端代码(如 `server.js`)也将被公开。例如：

```
// Anti-pattern. Don't do this or you'll be fired
app.use('/dist',
  express.static(path.resolve(PWD, 'build'))
)
```



这将使 `server.js` 暴露给攻击者，并且 `server.js` 可能包含秘密、API 密钥、密码以及 `/dist/server.js` 中的实现细节。

通过使用 `dist` 或 `public` 等子文件夹，仅将子文件夹暴露给用户(通过 HTTP)，并将前端文件放在这个暴露的子文件夹中，这样可以限制对其他文件的访问。

要使 GraphQL API 工作，需要设置路由(`/q`)，在路由中使用 `graphqlHTTP` 库以及模式(`server/schema.js`)和会话(`req.session`)来响应数据：

```
app.use('/q', graphqlHTTP(req => ({
  schema,
  context: req.session
})))
```

最后，为了使服务器正常工作，需要使服务器监听特定端口上的传入请求：

```
app.listen(PORT, () => console.log(`Running server on port ${PORT}`))
```

这里的 `PORT` 是一个环境变量。这个变量可以从命令行界面传入进程，如下所示：

```
PORT=3000 node ./build/server.js
```

### nodemon 对比 node

回想一下，在 `package.json` 中，使用 `nodemon`：

```
nodemon ./build/server.js
```

使用 `nodemon` 与运行 `node` 相同，但是如果对代码进行更改，`nodemon` 会重新启动代码。

**警告：**在第 14 章中，使用端口 8080，因为这是 Webpack 开发服务器的默认值。对于这个例子的 Express 服务器，使用 8080 没有任何问题，但是出于某些奇怪的历史原因，Express 应用默认运行在 3000 端口上。也许我们可以责怪 Rails！

服务器还使用以大写形式声明的另一个变量：`PWD`。它也是一个环境变量，但是它由 Node 设置为项目路径：`package.json` 文件所在文件夹的路径，也是项目的根文件夹。

最后，使用 `graphqlHTTP` 和 `schema` 变量。`graphqlHTTP` 来自 `express-graphql` 包，`schema` 是使用 GraphQL 定义构建的数据模式。

代码清单 15.2 是完整的服务器启动文件(`ch15/redux-graphql-netflix/ server/index.js`)。

### 代码清单 15.2 Express 服务器提供数据和静态资源服务

```
const path = require('path')
const express = require('express')
const graphqlHTTP = require('express-graphql')
const schema = require('./schema')
const {
  PORT = 3000,
  PWD = __dirname
} = process.env
const app = express()
```

保存此文件的工作目录(PWD为  
Print Working Directory的缩写)

```

app.use('/q', graphqlHTTP(req => ({
  schema,
  context: req.session
})))

app.use('/dist', express.static(path.resolve(PWD, 'build', 'public')))

app.use('*', (req, res) => {
  res.sendFile('index.html', {
    root: PWD
  })
})

app.listen(PORT, () =>,
  console.log(`Running server on port ${PORT}`))

```

在3000端口(不是8080端口)上启动  
服务

GraphQL 是强类型的，这意味着它使用/q 中的模式。正如你在项目结构中所见，在 server/schema.js 中定义了这个模式。接下来让我们看看数据是什么样的：数据的结构将决定将要使用的模式。

### 15.2.2 数据结构

这个应用是一个显示电影数据的 UI。因此，需要将相关数据存放在某处，最简单的方法是将数据保存在 JSON 文件(server/movies.json)中。该文件包含所有的电影，并且每一部电影都可以用一个具有一系列属性的普通对象来表示，因此整个文件是一个对象数组：

```

[
  {
    "title": "Pirates of the Caribbean: On Stranger Tides"
    ...
  }, {
    "title": "Pirates of the Caribbean: At World's End"
    ...
  }, {
    "title": "Avengers: Age of Ultron"
    ...
  }, {
    "title": "John Carter"
    ...
  }, {
    "title": "Tangled"
    ...
  }, {
    "title": "Spider-Man 3"
    ...
  }, {
    "title": "Harry Potter and the Half-Blood Prince"
    ...
  }, {
    "title": "Spectre"
    ...
  }, {

```

```
"title": "Avatar"
...
}, {
  "title": "The Dark Knight Rises"
  ...
}]
```

注意：该例使用的数据来自维基百科([https://en.wikipedia.org/wiki/List\\_of\\_most\\_expensive\\_films](https://en.wikipedia.org/wiki/List_of_most_expensive_films))的 10 部制作费用最高的电影。

每个对象都包含电影的名称、封面 URL、发行年份、以百万美元为单位的制作成本以及主演等信息。例如, *Pirates of the Caribbean*(《加勒比海盗》)的数据如下:

```
{
  "title": "Pirates of the Caribbean: On Stranger Tides",
  "cover": "/dist/images/On_Stranger_Tides_Poster.jpg",
  "year": "2011",
  "cost": 378.5,
  "starring": [{
    "name": "Johnny Depp"
  }, {
    "name": "Penélope Cruz"
  }, {
    "name": "Ian McShane"
  }, {
    "name": "Kevin R. McNally"
  }, {
    "name": "Geoffrey Rush"
  }]
}
```

目前每部电影都是只有名称的对象。之后可以添加任意数量的属性。但现在让我们来关注数据模式。

### 15.2.3 GraphQL 模式

可以在 GraphQL 中使用任何数据源: SQL 数据库、对象存储、文件或远程 API。但下面两件事情很重要:

- 数据纯度: 相同的请求应该有相同的响应(也就是说, 数据接口是幂等的)。
- 数据应该可以用 JSON 来表示。

导入存储在 JSON 文件中的电影列表:

```
const movies = require('./movies.json')
```

典型的 GraphQL 模式定义了一个带有字段和参数的查询。示例数据模式只有一个对象列表, 每个对象只有 title 属性。模式定义如下, 这是一个基本的例子, 稍后将看到完整的模式:



```

const movies = require('./movies.json')
new graphql.GraphQLSchema({
  query: new graphql.GraphQLObjectType({
    name: 'Query',
    fields: {
      movies: {
        type: new graphql.GraphQLList(new graphql.GraphQLObjectType({
          name: 'Movie',
          fields: {
            title: {
              type: graphql.GraphQLString
            }
          }
        })),
        resolve: () => movies
      }
    }
  })
})

```

从文件(模拟数据库)中导入电影列表数据

将模式中的title字段定义为字符串

定义查询的getter, 这个getter将发送JSON文件数据(也可以从数据库中获取)

核心思想是, 当执行查询时, 分配给 `resolve` 的函数会执行。在此之后, 只从这个函数的调用结果中选择要请求的对象的属性。这些属性在结果对象中, 而未列出的字段不会出现在结果对象中。因此, 需要在每次执行查询时指定要接收哪些属性。这种方式使的 API 更加灵活且高效: 可以根据需要在运行时安排部分数据。

该例有两种类型的查询和更多的字段。具体实现如代码清单 15.3 所示(ch15/redux-graphql-netflix/server/schema.js)。

代码清单15.3 GraphQL模式

```

const {
  GraphQLSchema,
  GraphQLObjectType,
  GraphQLList,
  GraphQLString,
  GraphQLInt,
  GraphQLFloat
} = require('graphql')
const movies = require('./movies.json')

const movie = new GraphQLObjectType({
  name: 'Movie',
  fields: {
    title: {
      type: GraphQLString
    },
    cover: {
      type: GraphQLString
    },
    year: {
      type: GraphQLString
    },
    cost: {
      type: GraphQLFloat
    }
  }
})

```

为所有字段定义适当类型

将对象的名称设置为“movie”, 以便可以在两个查询中使用

使用浮点类型

```

    starring: {
      type: new GraphQLList(new GraphQLObjectType({
        name: 'starring',
        fields: {
          name: {
            type: GraphQLString
          }
        }
      })),
    },
  },
})

module.exports = new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: {
      movies: {
        type: new GraphQLList(movie),
        resolve: () => movies
      },
      movie: {
        type: movie,
        args: {
          index: {
            type: GraphQLInt
          }
        },
        resolve: (r, {index}) => movies[index - 1]
      }
    }
  })
})

```

返回整个电影列表

使用索引(从URL参数中获取)返回单部影片信息

现在我们回到前端，看看如何查询这台简洁的小型服务器。

### 15.2.4 查询 API 并将响应保存到数据存储

要获取电影列表，需要查询服务器。前端收到服务器端的响应后，必须将响应的数据转交给数据存储。这个操作是异步执行的，而且涉及 HTTP 请求，所以是时候介绍 axios 了。

#### promise 和回调

axios 是基于 promise 的 HTTP 请求库。意思就是，在调用函数后立即返回一个 promise。因为 HTTP 请求不能保证立即执行，所以需要等到 promise 状态变成 resolved。

一旦 promise 状态变成 resolved，就可以使用它的 then() 方法。then() 方法接收一个回调函数，该函数接收操作结果为参数，在本例中参数为 HTTP 调用结果：

```

getPromise(options)
  .then((data) => {
    console.log(data)
  })

```

使用 promise 和回调(在 then() 方法中)可以替代仅仅使用回调的方式，如下是不使用



promise 重写的上述代码:

```
getResource(options, (data)=>{
  console.log(data)
})
```

有一个与 promise 相关的争议。尽管有些人喜欢 promise catch.all 语法的便利, 而不是简单的回调, 但其他人却觉得 promise 不值得那么麻烦(我也这么认为), 特别是考虑到 promise 可以掩盖错误, 并默默地失败。尽管如此, Promise 依然是 ES6/ES2015 的一部分, 并将继续保留。与此同时, 诸如 generator 和 async/await 的新模式正在成为编写异步代码的下一演进的一部分<sup>4</sup>。

请放心, 你可以继续使用普通回调进行异步编码。但是大多数现代(特别是前端)代码使用(或将使用)Promise 或 async/await。因此, 本书使用 fetch() 和 axios。

有关 Promise API 的更多信息, 请参阅 MDN(<http://mng.bz/7DcO>)上的文档和我的文章“Top 10 ES6 Features Every Busy JavaScript Developer Must Know” (<https://webapplog.com/es6>)。

axios 使用基于 promise 的请求, 与 fetch() 不同。要执行 GET HTTP 调用, 请使用 axios 的 get() 方法:

```
axios.get('/q')
```

由于 axios 返回一个 promise, 可以直接使用 then() 接收返回结果:

```
axios.get('/q').then(response => response)
```

为 then() 方法传递函数作为参数, 该函数被返回到 promise 的上下文中, 而不是返回到组件方法的上下文中。通过调用操作创建器将新数据发送到数据存储中:

```
axios.get('/q').then(response => this.props.fetchMovie(response))
```

现在, 让我们针对 GraphQL API 构建适当的查询。为此, 可以使用多行模板字符串(注意使用的是反引号而不是单引号):

```
axios.get(`/q?query={
  movie(index:1) {
    title,
    cover
  }
}`).then(response => this.props.fetchMovie(response))
```

使用多行模板字符串(反引号)保留换行符, 因此查询字符串会有新行。这样并不好, 因为查询字符串中的新行可能会中断 API 端点 URL。因此, 需要删除 HTTP 调用中不必要的空格和换行符, 但是在源代码中需要保留这些空格和换行符。clean-tagged-string 库(<https://github.com/taxigy/clean-tagged-string>)就只做了这么一件事——将一个巨大的多行模板字符串转换成一个较小的单行字符串, 例如

```
clean`/q?query={
```

4 请参阅我在 ES6 和 ES7+ES8 上的课程, 网址为 <https://node.university/p/es6> 和 <https://node.university/p/es7-es8>



```

movie(index:1) {
  title,
  cover
}
`

```

转换后的结果如下：

```

'/q?query={ movie(index:1) { title, cover } }'

```

注意语法：在 `clean` 之后没有圆括号，它被附加在模板字符串上。这是有效的语法，我们称这种语法为标记字符串(<http://mng.bz/9CqH>)。

现在，我们来看看索引为 1 的第一部电影：

```

const clean = require('clean-tagged-string').default
axios.get(clean`/q?query={
  movie(index:1) {
    title,
    cover
  }
}`).then(response => this.props.fetchMovie(response))

```

接下来，需要实现通过 `id` 获取电影信息的代码。需要请求更多的字段，而不仅仅是 `title` 和 `cover`，以便可以显示图 15.1 所示的视图。单个电影页面在重新加载时不会丢失，因为添加了特殊的服务器代码来捕获发送(`sendFile()`)`index.html` 文件的所有路由(\*)。

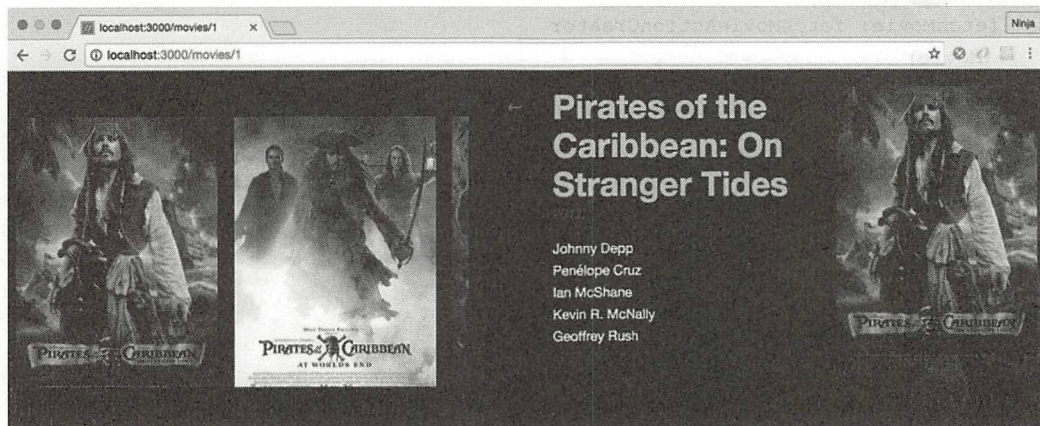


图 15.1 从使用浏览器记录(无哈希标记)的 Express 服务器(3000 端口)渲染的单个电影视图

可以使用你喜欢的基于 `promise` 的 HTTP 代理(`axios`)，在组件生命周期中通过 API 获取单部电影的数据：

```

componentWillMount() {
  const query = clean`{
    movie(index:${id}) {
      title,
      cover,
      year,

```

```

      starring {
        name
      }
    }
  }
}
axios.get(`/q?query=${query}`)
  .then(response =>
    this.props.fetchMovie(response)
  )
}

```

电影实体的请求属性列表要稍长一些，不仅仅是 `title` 和 `cover`，还包括 `year` 和 `starring`。因为 `starring` 本身是一个对象数组，所以还要声明需要请求的那些对象的属性，在本例中只需要 `name` 属性。

来自 API 的响应将流向 `fetchMovie` 操作创建器。之后，将使用用户希望看到的电影更新数据存储。

连接数据存储：

```

const {
  fetchMovieActionCreator
} = require('modules/movies.js')
...
module.exports = connect(({movies}) => ({
  movie: movies.current
})), {
  fetchMovie: fetchMovieActionCreator
})(Movie)

```

渲染页面：

```

render() {
  const {
    movie = {
      starring: []
    }
  } = this.props;
  return (
    <div>
      <img src={`url(${movie.cover})`} alt={movie.title} />
      <div>
        <div>{movie.title}</div>
        <div>{movie.year}</div>
        {movie.starring.map((actor = {}, index) => (
          <div key={index}>
            {actor.name}
          </div>
        ))}
      </div>
      <Link to="/movies">
        ?
      </Link>
    </div>
  )
}

```

```

    </Link>
  </div>
)
}

```

为了更好地组织代码,我们在第 14 章中你已经很熟悉的 `Movie` 组件中添加 `fetchMovie()` 方法(ch14/redux-netflix/src/components/movie/movie.js)。这个新方法用来做 AJAX 调用,然后分发操作。这个方法在 `Movie` 组件中,如代码清单 15.4 所示(ch15/redux-graphql-netflix/client/components/movie/movie.js)。

代码清单15.4 组件类方法`fetchMovie()`

```

// ...
fetchMovie(id = this.props.params.id) {
  const query = clean`
    movie(index:${id}) {
      title,
      cover,
      year,
      starring {
        name
      }
    }
  `
  axios.get(`/q?query=${query}`)
    .then(response => {
      this.props.fetchMovie(response)
    })
}
// ...

```

使用id、模板字符串和clean来创建查询

使用URL中的React Router参数来设置id

发送/q请求

使用服务器上的数据分发操作

接下来,我们继续获取电影列表。

### 15.2.5 显示电影列表

当显示电影列表时,对 API 的查询是不同的,而且渲染方式与获取单部电影的方式不同。使用有效的 GraphQL 查询从 GraphQL 服务器获取数据,通过使用 `axios` 库执行异步 GET 请求,然后将这些数据放到数据存储中。接下来要做的就是向用户显示这些数据:是时候渲染它们了。

通过前面章节你已经知道,为了从数据存储中获取数据,需要连接一个组件:使用 `connect()` 函数调用将状态和操作映射到属性。在组件的 `render()` 函数中,使用组件属性。但是这些属性需要值,这就是要进行 AJAX/XHR 调用的原因,这个调用通常是在组件第一次被挂载之后(生命周期事件)执行。

让我们声明一个从数据存储中选择数据的组件,该组件从属性中获取数据并进行渲染。首先,将组件连接到数据存储,以下代码片段来自 `ch15/redux-graphql-netflix/client/components/movies/movies.js`:



```
const React = require('react')
const { connect } = require('react-redux')
const {
  fetchMoviesActionCreator
} = require('modules/movies')
class Movies extends Component {
  // ...
}

module.exports = connect(({movies}) => ({
  movies: movies.all
}), {
  fetchMovies: fetchMoviesActionCreator
})(Movies)
```

`connect()` 函数有两个参数：第一个参数将数据存储映射到组件属性，第二个参数将操作创建器映射到组件属性。之后，组件会有两个新的属性：`this.props.movies` 和 `this.props.fetchMovies()`。

接下来，请求这些电影数据，并在收到数据时通过操作创建器将数据放入数据存储(分发操作)。通常，在组件生命周期 `componentWillMount()` 或 `componentDidMount()` 中请求远程 API 以获取数据：

```
const {
  fetchMoviesActionCreator
} = require('modules/movies.js')
...
class Movies extends React.Component {
  componentWillMount() {
    const query = clean`{
      movies {
        title,
        cover
      }
    }`

    axios.get(`/q?query=${query}`)
      .then(response => {
        this.props.fetchMovies(response)
      })
  }
  // ...
}

module.exports = connect(({movies}) => ({
  movies: movies.all
}), {
  fetchMovies: fetchMoviesActionCreator
})(Movies)
```

← 导入操作创建器

使用来自服务器的响应调度操作来更新数据存储

分发操作，这个操作由操作创建器提供

最后，使用属性中的数据渲染 `Movies` 组件，这些数据来自 `Redux` 数据存储：

```
// ...
render() {
  const {
    movies = []
```

```
} = this.props
return (
  <div>
    {movies.map((movie, index) => (
      <Link
        key={index}
        to={` /movies/${index + 1}`}>
        <img src={`url${movie.cover}`} alt={movie.title} />
      </Link>
    ))}
  </div>
)
// ...
```

每部电影都有 `cover` 和 `title` 属性。到电影的链接基本上是电影在 `movies` 数组中位置的索引。但是，当在一个集合中有成千上万个元素时，这种设置是不稳定的，因为不能保证顺序永远不变，不过现在没关系。更好的方法是使用唯一的 `id`，这个 `id` 通常由 MongoDB 这样的数据库自动生成。

该组件现在呈现电影列表，不过它还没有样式。查看本章的源代码，了解它如何与样式和三级层次结构的组件一起工作。

### 15.2.6 GraphQL 总结

添加基本级别的 GraphQL 支持是非常简单明了的。GraphQL 的工作方式与典型的 RESTful API 不同：因为前者的 API 提供任何实体子集，所以可以在任何嵌套级别查询任意属性。这使得 GraphQL 对于复杂对象的数据集更高效，而 REST 设计通常需要多个请求才能获得相同的数据。

GraphQL 是实现服务器-客户端握手的一种很有前途的模式。它允许客户端进行更多的控制，客户端可以将数据的结构指定给 REST API。这种控制反转允许前端开发者只请求他们需要的数据，而不必修改后端代码(或者要求后端团队修改代码)。

## 15.3 测验

1. 以下哪个命令可以创建 GraphQL 模式？`new graphql.GraphQLSchema()`、`new graphql.GraphQLSchema()` 还是 `new graphql.getGraphQLSchema()`？
2. 可以将 API 调用放入 `reducers` 中，这么理解正确还是错误？(提示：请参阅第 14 章的提示。)
3. 在哪里进行 GraphQL 调用来获取电影数据？`componentDidUnmount()`、`componentWillMount()` 还是 `componentDidMount()`？
4. ``/q?query=${query}`` 这种语法是什么？内联 Markdown、注释、模板文字、字符串模板还是字符串插值？

5. GraphQLString 是一种特殊的 GraphQL 模式类型, 可以从 graphql 包中取出这个类, 这么理解正确还是错误?

## 15.4 小结

- GraphQL 是一种向前端提供数据的健壮、可靠的方式, 并且消除了很多重复的后端代码。
- 要使用 React Router 启用浏览器记录和无哈希 URL, 可以在 Express\*路由中使用 `sendFile()` 发送 `index.html`。
- Express 不仅仅可以作为数据提供者, 还可以在 `app.use()` 中使用 `express.static`, 从而使 Express 作为静态 Web 服务器。
- GraphQL 的 URL 结构是 `/q?query = ...`, 其中的 `query` 具有数据查询的参数。

## 15.5 测验答案

1. `new graphql.GraphQLSchema()`
2. 错误。要避免将 API 调用放在 `reducer` 中。最好把它们放在组件中(容器/智能组件)。
3. `componentWillMount()`, 但 `componentDidMount()` 也可以。没有 `componentDidUnmount()` 这个方法。
4. 模板文字、字符串模板和字符串插值都可以。
5. 正确。这是有效的代码: `const {GraphQLString} = require('graphql')`, 见代码清单 15.3。



# 第 16 章

## 使用 Jest 进行单元测试

### 本章内容：

- 为什么使用 Jest
- 使用 Jest 进行单元测试
- 使用 Jest 和 TestUtils 进行 UI 测试

在现代软件工程中，测试很重要。它至少与使用敏捷方法、编写有良好文档的代码、保持工作热情一样重要，有时甚至比这些更重要。正确的测试可以节省很多调试时间。代码不是资产，而是负债，所以你的目标是使其维护工作尽可能简单。

### 为什么说代码是负债？

使用谷歌搜索短语“Code isn't an asset, it's a liability”，给出了大约 1.91 亿条结果，很难确定具体来源。我虽然找不到该短语的作者，但可以告诉你它的主旨：使用代码构建的应用/产品/服务是资产，但代码不是。资产可以产生收入，而代码本身并不产生任何收入。代码仅仅是制作产品(即资产)的工具。

代码是一种负债，因为你必须维护它们。更多的代码不会自动转化为更多的收入或更好的产品，反而总是会增加复杂性和维护成本。为了使维护成本降低，我们要尽可能使代码简单、稳健、灵活，以适应未来的变更和改进。而且当程序代码发生变更时，测试，特别是自动化测试，可以用来验证这些变更不会影响程序的正常运行。

使用测试驱动/行为驱动开发(Test-Driven/Behavior-Driven Development, TDD/BDD)可以使维护更容易。TDD/BDD 还可以让你更快速、更有效率地进行迭代开发，从而让你的公司更具竞争力。

## 16.1 测试的类型

有多种类型的测试。最常见的可以分为三类：单元测试、服务测试和 UI 测试，如图 16.1 所示。以下是对从最低到最高级别的每个测试类别的概述：

- 单元测试：系统要测试独立的方法和类。其中没有或很少有依赖关系或互连的部分。测试代码应足以验证被测试方法的工作原理。例如，生成随机密码的模块可以通过调用模块中的方法并将输出与正则表达式模式进行比较来测试。单元测试还可能涉及多个部分或模块协同以完成某个功能的测试。例如，几个组件必须协同工作，以提供带有强度检查的密码输入功能。可以通过将值提供给组件(输入)并监视强度检查的变化(无论是否满足)来进行测试。根据行业最佳实践经验，单元测试应占测试的大约 70%(见图 16.1)，应该绝对超过其他类型的测试。
- 服务(集成)测试：这类测试通常涉及其他依赖关系，需要单独的环境。集成测试应该占有所有测试的大约 20%。一旦有了单元测试的坚实基础和功能测试的保证，就不会希望有太多的集成测试，因为维护它们会导致开发缓慢。每次 UI 改变都有可能需要更新集成测试，导致 UI 测试非常混乱。
- UI(验收)测试：这类测试经常模拟用户操作，涉及整个系统，非常复杂，测试更脆弱、更难维护(成本高)，因此它们应该只占整个测试的 10%左右。

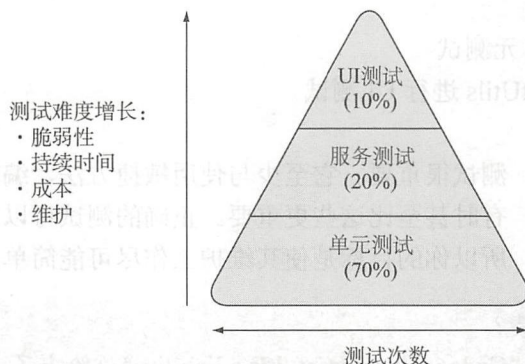


图 16.1 软件工程的最佳实践测试金字塔

本章介绍使用 React 和 Jest 的模拟 DOM 渲染对 React 组件进行的一些测试。你还将用到 Node、npm、Babel 和 Webpack 等标准工具链。

## 16.2 为什么使用 Jest(对比 Mocha)

Jest(<https://facebook.github.io/jest>)是基于 Jasmine 的命令行工具，它有一个茉莉花般的界面。如果使用过 Mocha，就会发现 Jest 看起来与之类似，很容易学习。Jest 由 Facebook 开发，经常与 React 一起使用；API 文档位于 <https://facebook.github.io/jest/docs/api.html#content>。

Jest 有如下特点：



- 强大的模拟 JavaScript/Node 模块的功能(<https://facebook.github.io/jest/docs/mock-functions.html>)使得更容易隔离代码以进行单元测试。
- 和其他测试框架相比, Jest 需要更少的设置, 例如 Mocha 需要导入 Chai 或其他独立的 Expect。Jest 的测试文件也在 `__tests__` 文件夹下。
- 测试被沙盒化, 并且可以并行执行, 以便更快地运行它们<sup>1</sup>。
- 可以在 Facebook 的 Flow(静态类型检查器, <https://flowtype.org>)的支持下执行静态分析。
- Jest 提供模块化、可配置性和适应性(通过 Jasmine 的断言来支持)。

### 模拟、静态分析与 Jasmine

术语“模拟”意味着伪造某个依赖关系的一部分, 以便可以测试当前代码。自动模拟意味着自动完成模拟。在 Jest v15<sup>2</sup>之前, 导入的每个依赖关系都被自动模拟, 如果经常需要模拟, 那么这个功能非常有用。但是大多数开发人员不需要自动模拟, 所以在 Jest v15+ 中, 自动模拟默认是关闭的, 但是如果需要, 可以启动自动模拟。

静态分析意味着代码可以在运行之前被分析, 通常涉及类型检查。Flow 是一个 JavaScript 静态类型检查库。

Jasmine 是一个带有断言语言的功能丰富的测试框架。Jest 建立在 Jasmine 引擎之上, 因此不需要导入或配置任何内容。可以轻松进入 Jasmine 的通用界面, 而无需额外的依赖关系或设置。

不同的测试框架适合不同的项目。大多数项目都在使用 Mocha, 不过 Jasmine 可以与 Mocha 和 Jest 相互交换。它们使用相同的结构定义测试套件:

- describe: 测试套件
- it: 测试用例
- before: 所有测试用例之前执行
- beforeEach: 每个测试用例之前执行
- after: 所有测试用例之后执行
- afterEach: 每个测试用例之后执行

在本书中我们不讨论哪些框架更优, 我们鼓励你保持开放的态度。你可以根据我列出的 Jest 特点, 以及 Jest 和 React 来自同一社区, 来更好地判断哪个框架更适用于你的下一个 React 项目。

总而言之, 大多数测试框架都是类似的, 可以根据个人风格及项目需求选择测试框架。Jest 是不错的选择, 一旦学会 Jest, 就可以随时更换为其他框架, 如 Mocha、Jasmine、node-tap。

## 16.3 使用 Jest 进行单元测试

如果没有用过其他测试框架, 那就直接学习 Jest。学习 Jest 时, 主要是学习 describe

<sup>1</sup> Christopher Poher, “JavaScript Unit Testing Performance,” *Jest*, March 11, 2016, <http://mng.bz/YfXz>

<sup>2</sup> Christoph Pojer, “Jest 15.0: New Defaults for Jest,” September 1, 2016, <http://mng.bz/p20n>



和 `it` 全局函数：`describe` 是一个测试套件，用作测试的包装器；`it` 是一个称为测试用例的单独测试。测试用例嵌套在测试套件中。

其他构造函数还有 `before`(在所有测试用例之前执行)、`after`(在所有测试用例之后执行)、`beforeEach`(在每个测试用例之前执行)和 `afterEach`(在每个测试用例之后执行)。

编写测试包括创建测试套件、用例和断言。断言就像真假问题，但使用的是可读性更好的格式(BDD)。

以下是一个不含断言的示例：

```

定义done回调
describe('Noun: method or a class/module name', () => {
  before((done) => {
    // This code will be called just once before all it statements
    done()
  })
  beforeEach((done) => {
    // This code will be called many times before all it statements
    done()
  })
  it('Verb describing the behavior', (done) => {
    // Assertions
    done()
  })
  it('Verb describing the behavior', (done) => {
    // Assertions
    done()
  })
  ...
  after((done) => {
    // This code will be called just once after all it statements
    done()
  })
  afterEach((done) => {
    // This code will be called many times after all it statements
    done()
  })
})

```

在异步测试代码执行完毕后调用done回调

测试必须至少包含一个 `describe` 和一个 `it`，其他的 `before`、`after` 等都是可选的。在测试 React 组件之前，你需要学习更多的知识。在本节中，我们创建一个生成随机密码的代码模块并测试它，这个模块可以自动生成随机密码，为简单起见，密码格式是 8 个字母数字组合。代码目录结构如下：

```

/generate-password
/ __test__
  generate-password.test.js
/node_modules
  generate-password.js
  package.json

```

使用 CommonJS/Node 模块语法，该语法在 Node 中得到了广泛支持，而且通过使用 Browserify 和 Webpack 使得该语法在浏览器中也得到了支持。代码清单 16.1 中是 `ch16/generate-password.js` 文件中的代码。

## 代码清单16.1 密码生成模块

```

module.exports = () => {
  return Math.random().toString(36).slice(-8)
}

```

使用带负数的slice()来反转次序(从右到左)

仅仅作为复习,在这个文件中,通过 `module.exports` 导出该函数。它是 Node.js 和 CommonJS 语法标记。可以通过使用 Webpack 和 Browserify(<http://browserify.org>)等额外工具使其可以在浏览器中运行。

该函数使用 `Math.random()` 生成一个数字,并将其转换成字符串,然后使用 `slice(-8)` 把它截取为 8 位字符。

要测试模块,可以从终端运行如下 Node 命令。该命令导入模块,并调用模块的方法,然后打印出结果:

```
node -e "console.log(require('./generate-password.js')())"
```

可以使用不同数目(而不仅仅是 8 个)的字体,改进这个模块。

## 16.3.1 在 Jest 中编写单元测试

在使用 Jest 前,需要新建项目目录,然后使用 `npm init` 初始化 `package.json` 文件。如果没有安装 npm,可以在附录 B 中查看安装方式。

在文件夹中创建完 `package.json` 文件后,使用如下命令安装 Jest:

```
$ npm install jest-cli@19.0.2 --save-dev --save-exact
```

本书使用 `jest-cli` 19.0.2 版本,建议你使用相同的版本。`--save-dev` 将 `jest-cli` 添加至 `package.json` 的 `devDependencies` 中。打开 `package.json`,然后手动将 `test` 脚本内容更改为 `jest`,如代码清单 16.2 所示(ch16/jest/package.json)。这将添加测试命令。另外还要添加启动脚本。

## 代码清单16.2 保存CLI测试脚本

```

{
  "name": "jest",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "jest",
    "start": "node -e
    ➔ \"console.log(require('./generate-password.js')())\""
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "devDependencies": {
    "jest-cli": "19.0.2"
  }
}

```

替换默认test脚本内容为jest

添加start脚本,执行Node命令以获取随机密码

去掉^符号,使其版本号固定为19.0.2

新建文件夹 `__tests__`，这个文件夹名很重要，因为 Jest 会从该文件夹中取出测试。在该文件夹中新建文件 `generate-password.test.js`。

通常情况下，只需要模拟依赖关系，而不需要隔离当前正在进行单元测试的库。

在 Jest v15 之前，会自动模拟每个依赖文件，你需要使用 `dontMock()` 或 `jest.autoMockOff()` 来避免模拟测试模块(`generate-password.js`)，例如：

```
jest.dontMock('../generate-password.js') // 在 v15 以前的 Jest 版本中生效
```

不过本书使用的是 v19，该版本已经默认禁止自动模拟功能。所以，你可以跳过 `dontMock()` 代码行或者把它注释掉。

测试文件中、含有一个测试套件(只有一个 `describe`)，用来测试生成的密码是否匹配正则表达式模式  `/^[a-z0-9]{8}$/ (只有数字和字母组合的 8 位字符) 以满足强密码的条件，参见代码清单 16.3(ch16/generate-password/__tests__/generate-password.test.js)。你肯定不希望你的聊天用户被暴力破解！`

代码清单 16.3 密码生成模块的测试文件 `generate-password.test.js`

```
describe('method generatePassword', () => {  
  let password  
  generatePassword = require('../generate-password')  
  it('returns a generated password of lower-case characters  
    and numbers with the length of 8', (done) => {  
    password = generatePassword()  
    expect(password).toMatch(/^[a-z0-9]{8}$/)  
    done()  
  })  
})
```

使用 Node.js 的全局 `require` 方法  
导入一个模块文件

如果定义了异步测试所需的参数和同步测试  
的可选参数(在本例中是同步测试)，就调用  
`done()`

使用 `$ npm test` 命令执行测试，将会看到如下输出信息：

```
Using Jest CLI v13.2.3, jasmine2  
PASS __tests__/generate-password.test.js (0.031s)  
1 test passed (1 total in 1 test suite, run time 1.339s)
```

显示了有多少个测试套件通过，  
以及一共有多少个测试套件

## 16.3.2 Jest 断言

默认情况下，Jest 使用由 Expect 语法(<https://facebook.github.io/jest/docs/api.html>)驱动的 BDD([https://en.wikipedia.org/wiki/Behavior-driven\\_development](https://en.wikipedia.org/wiki/Behavior-driven_development))语法。Expect 是一种流行的语言，是 TDD 断言的替代品。它有多种风格，Jest 使用了一个稍微简化的版本(在我看来)。不像其他的框架(如 Mocha)需要安装额外的语法支持模块，在 Jest 中已经默认内置了断言支持。



## TDD 和 BDD

TDD 可以指测试驱动开发或是带有断言的 TDD 语法。简单地说,在测试驱动开发过程中,编写一个测试,然后运行它(失败),让它工作(通过),最后让它正确(重构)。

当然可以使用 BDD 进行测试驱动开发。BDD 风格的主要优势在于它的目的是与跨功能团队的每个成员进行交流,而不仅仅是软件工程师。TDD 更多的是一种技术语言。BDD 格式的测试更加易读、可理解——理想情况下,规范的标题应该告诉你正在测试什么,就像在下面这个例子中一样:

```
describe('method generatePassword', ()=>{
  ...
  it('returns a generated password of lower-case characters
    and numbers with the length of 8', ()=>{
    ...
    expect(password).toMatch(/^[a-z0-9]{8}$/)
  })
})
```

使用名词描述测试套件

使用动词描述测试用例的行为

使用期望语句实现测试用例

以下是 Jest 支持的主要 Expect 方法列表(还有更多)。将程序返回的实际值传递给 `expect()`, 并使用以下方法将这些值与测试中硬编码的期望值进行比较:

- `.not`: 对之后的断言取反
- `expect(OBJECT).toBe(value)`: 断言目标严格等于 `(===)value`<sup>3</sup>
- `expect(OBJECT).toEqual(value)`: 断言目标深度等于 `value`<sup>4</sup>
- `expect(OBJECT).toBeFalsy()`: 断言目标为假值
- `expect(OBJECT).toBeTruthy()`: 断言目标为真值
- `expect(OBJECT).toBeNull()`: 断言目标为 `null`
- `expect(OBJECT).toBeUndefined()`: 断言目标为未定义
- `expect(OBJECT).toBeDefined()`: 断言目标为已定义
- `expect(OBJECT).toMatch(regexp)`: 断言目标匹配正则表达式 `regexp`

## 真值和假值

在 JavaScript/Node 中,当在 `if/else` 语句中作为布尔值进行求值时,真值将被转换为 `true`,假值将被转换为 `else`。

根据官方提供,仅有如下 6 个值是假值:

- `false`
- `0`
- `""`(空字符串)
- `null`
- `undefined`

3 “Equality Comparisons and Sameness”, Mozilla Developer Network, <http://mng.bz/kliO>

4 深度相等指的是对象所有层级的属性和值均相等,标准 JavaScripts 没有提供相关 API,可以使用第三方库来做深度比较,如 `deep-equal`([www.npmjs.com/package/deep-equal](http://www.npmjs.com/package/deep-equal))



- NaN

除此以外，其他值都是真值。

总而言之，Jest 可以用于单元测试，而且应该是做得最多的测试。它们比较底层，因此它们更坚固、更不易碎，这使得维护它们的成本更低。

到目前为止，你已经创建了一个模块，并使用 Jest 测试了它的方法。这是典型的单元测试，只有被测试的模块本身，没有涉及的依赖关系。这个技能应该让你准备好继续测试 React 组件了。接下来，我们来看看更复杂的 UI 测试。以下内容介绍 React 测试工具，我们将用它执行 UI 测试。

## 16.4 使用 Jest 和 TestUtils 进行 React UI 测试

一般来说，在 UI 测试(建议只占 10%的测试)中，可以测试整个组件、组件的行为，甚至整个 DOM 树。可以手动测试组件，但我们不建议这么做，手工测试需要很长时间，而且不是很可靠，所以应该尽量最小化或不存在手动 UI 测试。

自动化 UI 测试怎么样？可以使用无头浏览器([https://en.wikipedia.org/wiki/Headless\\_browser](https://en.wikipedia.org/wiki/Headless_browser))进行自动测试，无头浏览器类似于真正的浏览器，但没有 GUI。大多数 Angular 1 应用程序就是使用这种方式进行测试的。当然，React 也可以使用这种方式进行测试，但这并不是件容易的事，过程通常很慢，并且需要大量的处理能力。

另一种自动化 UI 测试的方法就是使用 React 的虚拟 DOM，可以通过 jsdom(<https://github.com/tmpvar/jsdom>)实现的类似浏览器的 JavaScript 环境进行访问测试。要使用 React 的虚拟 DOM，需要与 React 核心库密切相关的实用程序 TestUtils，这是一个用于测试 React 组件的应用程序。简单地说，TestUtils 允许创建组件并将其渲染到虚拟 DOM 中，然后可以查看标签和元素。所有这些都是从命令行完成的，并不需要浏览器(包括无头浏览器)。

注意：React 在 <https://facebook.github.io/reactions/docs/addons.html> 上列出了一些附加功能，这些功能大多不再推荐使用或者处在试验阶段，意味着 React 团队可能会更改使用方式或停止支持。这些附加功能都遵循 react-addons-NAME 命名规范。TestUtils 也是附加组件，像其他附加组件一样需要通过 npm 安装(如果没有 npm，将不能使用 TestUtils，npm 的安装方式详见附录 A)。

在 React v15.5.4 之前的版本中，TestUtils 的 npm 包名为 react-addons-test-utils (<https://facebook.github.io/react/docs/test-utils.html>)。例如，使用如下命令安装 React v15.2.1 相应的 TestUtils：

```
$ npm install react-addons-test-utils@15.2.1 --save-dev --save-exact
```

在 React v15.5.4 之前通过如下方式使用 TestUtils：

```
const TestUtils = require('react-addons-test-utils')
```

在 React v15.5.4 及之后版本中，TestUtils 已被打包到 react-dom 中，因为我们使用的就是 15.5.4 版本，所以无须单独安装，直接使用如下方式引入：

```
const TestUtils = require('react-dom/test-utils')
```

TestUtils 有几种渲染组件的主要方法；可以模拟 click、mouseover 等事件；并在渲染组件中查找元素。我们首先渲染一个组件，并在需要时再去了解其他方法。

为了说明 TestUtils 的 render() 方法，代码清单 16.4 在不使用浏览器的情况下将元素插入到 div 元素中(ch16/testutils/\_test\_/render:props.js)。

#### 代码清单16.4 使用Jest渲染React元素

```
describe('HelloWorld', () => {
  const TestUtils = require('react-dom/test-utils')
  const React = require('react')

  it('has props', (done) => {
    class HelloWorld extends React.Component {
      render() {
        return <div>{this.props.children}</div>
      }
    }

    let hello = TestUtils.renderIntoDocument(<HelloWorld>Hello Node!
    </HelloWorld>)

    expect(hello.props).toBeDefined()

    console.log('my hello props:', hello.props) // my div: Hello Node!

    done()
  })
})
```

在第 16 章所示示例的 package.json 中，添加 Babel、jest-cli、react、react-dom 等，使其看起来如下：

```
{
  "name": "password",
  "version": "2.0.0",
  "description": "",
  "main": "index.html",
  "scripts": {
    "test": "jest",
    "test-watch": "jest --watch",
    "build-watch": "./node_modules/.bin/webpack -w",
    "build": "./node_modules/.bin/webpack"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"
    ]
  },
}
```



```

    "devDependencies": {
      "babel-jest": "19.0.0",
      "babel-preset-react": "6.24.1",
      "jest-cli": "19.0.2",
      "react": "15.5.4",
      "react-dom": "15.5.4"
    }
  }
}

```

**警告：** `renderIntoDocument()` 仅适用于自定义组件，对于标准 DOM 组件(例如 `<p>`, `<div>`, `<section>`等)并不适用。如果看到“Error: Invariant Violation: findAllInRenderedTree(...): instance must be a composite component”错误提醒，确保使用的是自定义组件类而不是标准类。更多信息请查看 <http://mng.bz/8AOc> 和 <https://github.com/facebook/react/issues/4692>。

渲染组件 `hello` 后，其中 `hello` 组件具有 React 组件树(包括所有子组件)的值，可以使用任何一种元素查找方法获取组件及子组件的元素。例如，使用代码清单 16.5 所示的方式获取组件 `<HelloWorld>` 内的 `<div>` 元素(`ch16/testutils/__tests__/scry-div.js`)。

#### 代码清单 16.5 查找组件内的 `<div>` 元素

```

describe('HelloWorld', () => {
  const TestUtils = require('react-dom/test-utils')
  const React = require('react')

  it('has a div', (done) => {
    class HelloWorld extends React.Component {
      render() {
        return <div>{this.props.children}</div>
      }
    }

    let hello = TestUtils.renderIntoDocument(
      <HelloWorld>Hello Node!</HelloWorld>
    )
    expect(TestUtils.scryRenderedDOMComponentsWithTag(
      hello, 'div'
    ).length).toBe(1)
    console.log('found this many divs: ',
      TestUtils.scryRenderedDOMComponentsWithTag(hello, 'div').length)
    done()
  })
})
...

```

`scryRenderedDOMComponentsWithTag()` 允许通过标签名称(如 `div`)获取元素数组。当然还有其他获取元素的方法。

#### 16.4.1 使用 `TestUtils` 查找元素

除了 `scryRenderedDOMComponentsWithTag()` 之外，还有一些其他的方法用来获取元素列表(`scry` 为前缀，表示多组件)或单个元素(`find` 为前缀，表示单个组件)。两者都使用元素

类而不是组件类。

除了标签名称之外，还可以通过类型(组件类)或 CSS 类获取元素。例如，HelloWorld 是一种类型，而 div 是标签名称。可以组合使用如下 6 个方法来获取想要的元素：

- `scryRenderedDOMComponentsWithTag()`：通过元素类型返回元素列表。
- `findRenderedDOMComponentWithTag()`：通过元素类型返回单个元素，在组件内该元素必须唯一。
- `scryRenderedDOMComponentsWithClass()`：通过类名返回元素列表。
- `findRenderedDOMComponentWithClass()`：通过类名返回单个元素，在组件内该类名必须唯一。
- `scryRenderedComponentsWithType()`：通过自定义组件类型返回元素列表。
- `findRenderedComponentWithType()`：通过自定义组件类型返回单个元素，在组件内该组件类型必须唯一。

正如你所看到的，我们并不缺少从组件中提取必要元素的方法。如果需要一些指导，建议使用类或类型(组件类)，因为它们让你的目标元素更加健壮。例如，假设现在使用标签名称，因为只有一个<div>。现在你决定将具有相同标记名的元素添加到代码中(多个<div>)，这时就不得不重新编写测试。如果使用 HTML 类来测试<div>，那么将更多<div>元素添加到组件后，之前的测试依然可以正常工作。

使用标签名称的唯一情况可能是合适的，就是当需要使用特定标签名称(`scryRenderedDOMComponentsWithTag()`)测试所有元素，或者组件太小以至于没有其他具有相同标签名称(`findRenderedDOMComponentWithTag()`)的元素时。例如，如果有一个包含锚点标签<a>的无状态组件，并且向其中添加了一些 HTML 类，就不会有其他锚点标签。

只有在需要测试具有特定标签名称的所有元素时才推荐使用 `scryRenderedDOMComponentsWithTag()`，或是组件足够小，并确保组件内不会出现相同的标签名称时才使用 `findRenderedDOMComponentWithTag()`。例如，如果有一个包含锚点标签<a>的无状态组件，并且在组件中添加了其他一些 HTML 类，那么将没有额外的锚点标签。

## 16.4.2 UI 测试密码部件

现在考虑注册页面上使用的 UI 小部件，该部件用来自动生成一定强度的密码。如图 16.2 所示，它有一个输入框、一个 Generate 按钮和一个标准列表。

以下部分将介绍整个项目。现在，我们专注于使用 TestUtils 及其界面。一旦安装 TestUtils 和其他依赖(如 Jest)，就可以在目录 password/\_\_tests\_\_ 下创建 Jest 测试文件 password.test.js 对你的小部件进行 UI 测试，测试代码的结构如下：

```
describe('Password', function() {  
  it('changes after clicking the Generate button', (done)=>{  
    // Importations  
    // Perform rendering  
    // Perform assertions on content and behavior  
    done()  
  })  
})
```

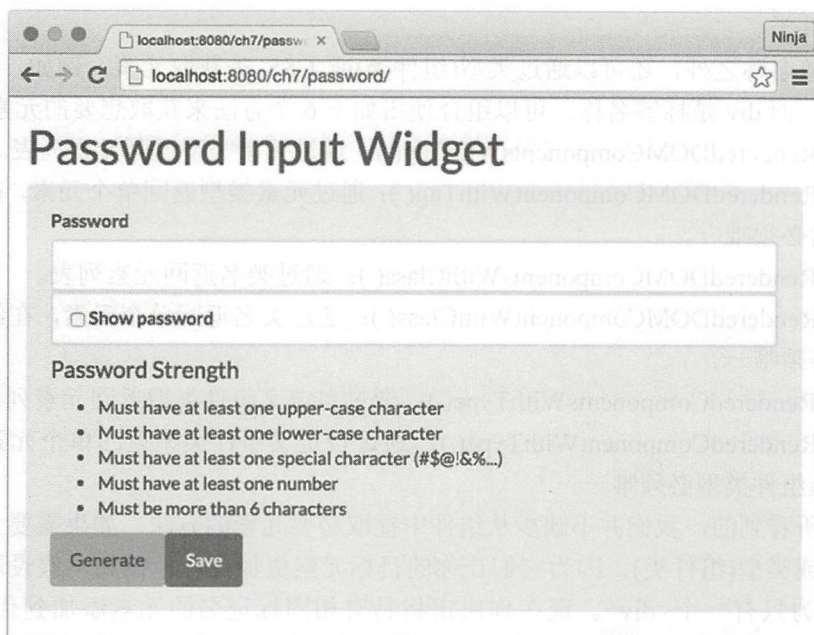


图 16.2 根据给定的强度标准自动生成密码的密码小部件

定义 `describe` 中的依赖关系，请注意我将 `ReactDOM.findDOMNode` 赋值给常量 `fd`，因为你将经常使用它：

```
const TestUtils = require('react-dom/test-utils')
const React = require('react')
const ReactDOM = require('react-dom')
const Password = require('../jsx/password.jsx')
const fd = ReactDOM.findDOMNode
```

使用 `renderIntoDocument()` 渲染组件。例如，渲染 `Password` 组件并保存引用到变量 `password` 中。传递的属性是密码强度规则的键(key)。例如，`upperCase` 的意思是至少一个大写字符：

```
let password = TestUtils.renderIntoDocument(<Password
  upperCase={true}
  lowerCase={true}
  special={true}
  number={true}
  over6={true}
/>
)
```

在本例中我们使用 JSX 语法，因为当安装了 `babel-jest` 模块(`npm i babel-jest --save-dev`)，并且配置 Babel 的预设为 `["react"]` 时，Jest 会自动使用 `babel-jest`。如果不引入 `babel-jest`，就不能使用 JSX，但可以使用 `createElement()`：

```
let password = TestUtils.renderIntoDocument(
  React.createElement(Password, {
    upperCase: true,
```



```

    lowerCase: true,
    special: true,
    number: true,
    over6: true
  })
)

```

使用 `renderIntoDocument()` 渲染组件后，可以直接提取所需的元素(Password 的子元素)，并执行断言来查看窗口小部件的工作原理。将提取调用想象成 jQuery，可以使用标签或类。我们最少应该测试如下事项：

- 有一个 Password 元素，其中包含作为强度标准的列表(<li>)。
- 强度列表的第一条有特定的文本。
- 强度列表的第二条未满足(删除线)。
- 有一个可单击的 Generate 按钮(类名为 generate-btn)，并单击这个按钮。
- 单击 Generate 按钮后，强度列表的第二条用删除线显示。

单击 Generate 按钮，生成满足所有标准的密码并使其可见(用户可以查看并记住)，本书中没有该功能的测试代码，将此任务作为课后作业。

让我们从检查第 1 项开始：TestUtils.scrRenderedDOMComponentsWithTag() 从一个特定的标签获取所有元素。在本例中，标签是 li，因为我们使用<ul> <li>来表示列表。toBe() 的作用类似于全等号(===)，可用于验证列表长度是否为 5：

```

let rules = TestUtils.scrRenderedDOMComponentsWithTag(password, 'li')
expect(rules.length).toBe(5)

```

检查第 2 项：使用 toEqual() 测试强度列表中的第一条是特定文本，例如期望列表的第一条文本为至少有一个大写字母，这也是密码强度的一条规则：

```

expect(fd(rules[0]).textContent).toEqual('Must have
  ➡ at least one uppercase character')

```

检查第 3 至第 5 项：找到一个按钮，并单击它，然后校验强度列表中第二条的值(必须从文本更改为删除线)。

### toBe() 对比 toEqual()

toBe() 和 toEqual() 在 Jest 中是不一样的，它们的行为有所不同。记住它们区别的最简单方法是：toBe() 是全等比较(===)，toEqual() 是深度比较所有属性和值。因此，如下两个断言都为真：

```

const copy1 = {
  name: 'React Quickly',
  chapters: 19,
}
const copy2 = {
  name: 'React Quickly',
  chapters: 19,
}

describe('Two copies of my books', () => {

```

```

it('have all the same properties', () => {
  expect(copy1).toEqual(copy2) // correct
})
it('are not the same object', () => {
  expect(copy1).not.toBe(copy2) // correct
})
})

```

但是当比较数字 5 或字符串 “Must have at least one uppercase character,” 时，两者会产生相同的结果：

```

expect(rules.length).toBe(5) // correct
expect(rules.length).toEqual(5) // correct
expect(fD(rules[0]).textContent).toEqual('Must have
➡ at least one upper-case character') // correct
expect(fD(rules[0]).textContent).toBe('Must have
➡ at least one upper-case character') // correct

```

有一个与 `TestUtils.scrRenderedDOMComponentsWithTag()` 类似的 `TestUtils.findRenderedDOMComponentWithClass()` 方法，但该方法只返回一个元素；如果有多个元素，则会抛出错误。为了模拟用户操作，还有一个 `TestUtils.Simulate` 对象，该对象含有的方法是驼峰命名格式的事件名称，例如 `Simulate.click`、`Simulate.keyDown` 和 `Simulate.change`。

我们使用 `findRenderedDOMComponentWithClass()` 获取按钮，然后使用 `Simulate.click` 单击它，所有这些都是在没有浏览器的代码中完成的：

```

let generateButton =
  ➡ TestUtils.findRenderedDOMComponentWithClass(password, 'generate-btn')
  expect(fD(rules[1]).firstChild.nodeName.toLowerCase()).toBe('#text')
  TestUtils.Simulate.click(fD(generateButton))
  expect(fD(rules[1]).firstChild.nodeName.toLowerCase()).toBe('strike')

```

该测试检查 `<li>` 组件在单击按钮时是否有 `<strike>` 元素(为文本加删除线)。单击按钮将生成一个随机密码，该密码满足第二个条件(`rules[1]`)，至少包含一个小写字母以及其他条件。到这里就已经完成了该测试，让我们继续下一个测试。

你已经看到了 `TestUtils.Simulate` 的作用。它不仅可以触发单击操作，还可以触发其他交互，例如改变输入字段中的值或触发回车键(`keyCode 13`)：

```

ReactTestUtils.Simulate.change(node)
ReactTestUtils.Simulate.keyDown(node, {
  key: "Enter",
  keyCode: 13,
  which: 13})

```

注意：因为 `TestUtils` 不会自动生成组件中所需的数据，所以必须手动传递在组件中使用的数据，例如 `key` 或 `keyCode`。React 支持的每个用户操作在 `TestUtils` 中都有相应的方法。

可以参考项目清单文件 `package.json`，该文件还包括下面将介绍的浅渲染(`shallow rendering`)库。如果要运行 `ch16/password` 示例，需要使用 `npm i` 安装依赖，然后执行 `npm test`：



```
{
  "name": "password",
  "version": "2.0.0",
  "description": "",
  "main": "index.html",
  "scripts": {
    "test": "jest",
    "test-watch": "jest --watch",
    "build-watch": "./node_modules/.bin/webpack -w",
    "build": "./node_modules/.bin/webpack"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"
    ]
  },
  "devDependencies": {
    "babel-core": "6.10.4",
    "babel-jest": "13.2.2",
    "babel-loader": "6.4.1",
    "babel-preset-react": "6.5.0",
    "jest-cli": "19.0.2",
    "react": "15.5.4",
    "react-dom": "15.5.4",
    "react-test-renderer": "15.5.4",
    "webpack": "2.4.1"
  }
}
```

接下来，我们来看看渲染 React 元素的另一种方法。

### 16.4.3 浅渲染

在某些情况下，可能仅需要测试单个级别的渲染，即组件中的 `render()` 结果并不包含组件的子组件(如果有的话)。这样可以简化测试，因为不需要 DOM 环境(系统创建的元素)就可以断言它的事实。使用前我们需要安装 `react-test-renderer v15.5.4` 包(如果使用的是更早版本的 React，那么 `react-test-renderer` 是集成在 `TestUtils` 中的)：

```
npm i react-test-renderer -SE
```

为了说明，下面用浅渲染方法测试相同的密码元素。如下代码可以在相同的测试文件 `ch16/password/__tests__/password.test.js` 中运行。在本例中，创建一个渲染器，然后传递组件给该渲染器并获得组件的浅渲染结果：



```
const { createRenderer } = require('react-test-renderer/shallow')
const passwordRenderer = createRenderer()
passwordRenderer.render(<Password/>)
let p = passwordRenderer.getRenderOutput()
expect(p.type).toBe('div')
expect(p.props.children.length).toBe(6)
```

对浅渲染的结果执行断言

执行浅渲染

现在，如果在 `console.log(p)` 中记录 `p`，结果将包含子对象，但对象 `p` 不是 React 实例。浅渲染的结果如下：

```
{ '$$typeof': Symbol(react.element),
  type: 'div',
  key: null,
  ref: null,
  props:
    { className: 'well form-group col-md-6',
      children: [ [Object], [Object], [Object], [Object],
        [Object], [Object] ] },
  _owner: null,
  _store: {} }
```

与 `renderIntoDocument(<Password/>)` 的结果日志相比，后者生成带有状态的 `Password` (React 元素) 实例。下面是完整的渲染结果(非浅渲染)：

```
Password {
  props: {},
  context: {},
  refs: {},
  updater:
    { ...
    },
  state: { strength: {}, password: '',
    visible: false, ok: false },
  generate: [Function: bound generate],
  checkStrength: [Function: bound checkStrength],
  toggleVisibility: [Function: bound toggleVisibility],
  _reactInternalInstance:
    { _currentElement:
      { '$$typeof': Symbol(react.element),
        type: [Function: Password],
        key: null,
        ref: null,
        props: {},
        _owner: null,
        _store: {} },
      ...
    }
  }
```

有state值(浅渲染没有)

元素值很像浅渲染结果

很明显，我们无法使用浅渲染来测试用户行为和嵌套元素，但是浅渲染可用于测试组件中第一级的子级以及组件的类型，可以将此功能用于测试自定义(可组合的)组件类。

在实际使用中，可以使用浅渲染来对单个组件及其渲染进行高针对性(很像单元测试)

的测试，当不需要测试子组件、用户行为或更改组件的状态时，换句话说，当只需要测试单个元素的 `render()` 函数时，就可以使用它。根据经验，首先使用浅渲染，如果浅渲染无法满足需求，再使用正常的渲染进行测试。

标准的 HTML 类可以检查和断言 `el.props`，所以不需要浅渲染。例如，创建一个锚点元素并测试它是否具有预期的类名和标签名称：

```
let el = <a className='btn' />
expect(el.props.className).toBe('btn')
expect(el.type).toBe('a')
```

## 16.5 TestUtils 总结

现在你已经学习了很多关于 TestUtils 和 Jest 的知识，足以供你在项目中使用。正如你在本书第 II 部分所做的工作：使用 Jest 和 TestUtils 进行 React 组件的行为驱动开发(第 18 至第 20 章)。如果想查看 Webpack 的配置以及实际项目中使用的所有依赖关系，可查看第 19 章的密码小部件。

有关 TestUtils 的更多信息，请参阅 <https://facebook.github.io/react/docs/test-utils.html> 上的官方文档。Jest 是一个广泛的话题，完全覆盖超出了本书的范围。请随时查阅官方 API 文档以了解更多信息：<https://facebook.github.io/jest/docs/api.html#content>。

最后，Enzyme 库(<https://github.com/airbnb/enzyme> 和 <http://mng.bz/Uy4H>)提供了相比 TestUtils 更多的功能和方法以及更加紧凑的方法名称。该库由 Airbnb 开发，并且需要依赖 TestUtils 以及 jsdom(它与 Jest 一起出现，所以只有在不使用 Jest 的情况下才需要 jsdom)。

测试是一件烦琐的事，有些开发人员会跳过测试，这太可怕了，不应该这么做。如果坚持下来，那么恭喜你，你的代码质量将会更好，也会开发更快，生活也会更幸福。不必半夜醒来去修复损坏的服务器，至少不会像那些没有测试的人那么频繁。

## 16.6 测验

1. Jest 测试必须位于以下哪个文件夹中？`tests`、`__test__` 还是 `__tests__`？
2. TestUtils 使用 npm 形式 `react-addons-test-utils` 来安装，这么理解正确还是错误？
3. TestUtils 的哪个方法可以通过类查找单个 HTML 元素？
4. 深度比较两个对象的方法是什么？
5. 如何测试用户的鼠标悬停行为？

## 16.7 小结

- 使用 `npm i jest-cli --save-dev` 安装 Jest。
- 使用 `jest.dontMock()` 关闭模块自动模拟。
- 使用 `expect.toBe()` 及其他 Expect 函数。

- 使用 `npm i react-addons-test-utils --save-dev` 安装 TestUtils。
- 使用 `TestUtils.Simulate.eventName(node)` 测试模拟 DOM 事件。
- 使用 `scry...` 方法获取元素列表。
- 使用 `find...` 方法获取单个元素(有零个或两个及以上匹配元素时会报错)。

## 16.8 测验答案

1. `__tests__`。这是 Jest 遵循的约定。
2. 正确。TestUtils 是一个单独的 npm 模块。
3. `findRenderedDOMComponentWithClass()`。
4. `expect(OBJECT).toEqual(value)`。
5. `TestUtils.Simulate.mouseOver(node)`。鼠标悬停事件通过悬停光标来触发。



# 第 17 章

## 在 Node 中使用 React 和同构 JavaScript

本章内容：

- 在服务器端使用 React
- 理解同构 JavaScript
- 在 Node 中使用 React
- 使用 React 和 Express
- 使用 Express 和 React 的同构 JavaScript

React 主要是一个前端库，用于在浏览器上构建完整的单页面应用或简单的 UI。那么我们为什么要关心在服务器上使用它？是不是在服务器上渲染 HTML 的老办法？是，也不是。事实证明，当在浏览器上构建 Web 应用时，会错过几个关键的好处。比如，错过谷歌搜索结果中排名靠前的位置，甚至可能损失数百万美元的收入。

继续阅读，找出原因。只有一种情况可以跳过本章：还没有注意到应用的性能(也就是说，你是开发新手)。其他人请继续阅读本章内容，你将获得宝贵的知识，可以用来构建令人惊叹的应用，并让你在开发者狂欢派对中使用“同构 JavaScript”这个术语时显得更聪明些。还将学习如何在 Node 中使用 React 和构建 Node 服务器，到本章末尾，你将了解如何使用 React 和 Express 构建同构 JavaScript 应用。

提示：如果之前没有了解过 Express，请阅读我的著作 *Pro Express.js* (Apress, 2014)，该书涵盖了 Express 第 4 版的内容。也可以阅读 Evan Hahn 的著作 *Express in Action* (Manning, 2015)。还可以查看我的在线课程 *Express Foundation*：<https://node.university/p/express->

foundation。如果熟悉 Express，但需要进一步了解，可以在附录 C 中找到 Express.js 的速查表，并且附录 A 介绍了如何安装 Express。

## 17.1 为什么在服务器端使用 React？什么是同构 JavaScript？

你可能已经听说过有关 Web 开发的同构 JavaScript。这已经成为一个流行语，似乎 2016 年的每一场 Web 技术大会都会不止一次地介绍同构 JavaScript。通用 JavaScript 甚至有一些同义词，比如同构 JavaScript 和全栈 JavaScript。为了简单起见，本章将一直使用同构 JavaScript。本节将帮助你了解什么是同构 JavaScript。

但在定义同构 JavaScript 之前，让我们先来讨论构建 SPA 应用时面临的一些问题。其中三个主要问题是：

- 没有搜索引擎优化(Search Engine Optimization, SEO)：单页面应用(SPA)完全在浏览器上生成 HTML，而搜索爬虫并不喜欢这样做。
- 性能差：巨大的打包文件和 AJAX 调用会降低性能(特别是在第一次加载页面时，这一点很关键)。
- 可维护性差：通常，SPA 应用会导致浏览器和服务器上的代码重复。

让我们仔细看看这些问题。

### 17.1.1 正确的页面索引

使用 Backbone、Angular、Ember 等框架构建的 SPA 应用被广泛用于受保护的应用——需要用户输入用户名和密码才能访问的应用(见图 17.1)。大多数 SPA 应用为受保护的资源提供服务，并不需要建立索引，但绝大多数网站都不受保护。

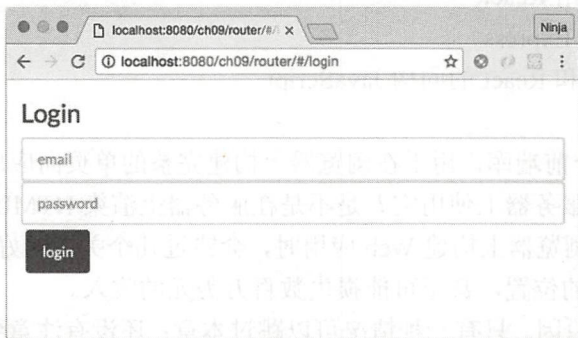


图 17.1 不需要 SEO 支持的 SPA 应用，因为需要登录才能访问

对于这样的公共应用，搜索引擎优化是重要的和强制性的，因为它们的业务严重依赖搜索索引和自然流量，并且大部分网站都属于这一类。

遗憾的是，当尝试将 SPA 架构应用于面向公众的网站时，并不能直接提供良好的搜索引擎索引。SPA 应用依赖浏览器渲染，因此需要重新实现服务器上的模板，或者使用无头浏览器为搜索引擎爬虫预生成静态 HTML 页面。



### Google 支持浏览器渲染

最近, Google 为其抓取工具添加了 JavaScript 渲染功能。你可能会认为, 这意味着现在浏览器渲染的 HTML 将被正确编入索引。你可能还会认为, 通过使用带有 REST API 服务器的 Angular, 不再需要服务器端渲染。遗憾的是, 情况可能并非如此。

以下内容来自 Google 网站管理员中心的博客文章“更好地理解网页”(http://mng.bz/Yv3B): “有时, 渲染过程中的事情并不完美, 这可能会对网站的搜索结果产生负面影响。”要点是 Google 不建议我们依靠它对 SPA 应用做索引。Google 无法保证缓存、索引和搜索结果中的内容正是 SPA 所渲染的内容。所以, 为了安全起见, 需要让不使用 JavaScript 和使用 JavaScript 的渲染结果尽可能一致。

通过同构 JavaScript 和 React, 特别是可以在服务器上使用相同的组件(浏览器使用这些组件生成 HTML 并呈现给用户)生成 HTML 并提供给爬虫, 不需要庞大的无头浏览器就能在服务器上生成 HTML。

### 17.1.2 更快的加载速度、更好的性能

虽然一些应用必须具有适当的搜索引擎索引, 但其他应用会在更高的性能下发展更快。研究表明: 像 <http://mobile.walmart.com><sup>1</sup>和 <http://twitter.com><sup>2</sup>这样的网站需要在服务器上渲染第一页(第一次加载)来提高性能。如果第一页的加载速度不够快, 用户会离开, 公司也会因此损失数百万美元。

作为一名 Web 开发者, 一般在网络较好的环境下工作和生活, 很有可能会忽略网站在慢速网络下的访问情况。在一些情况下, 本身 1 秒钟可以加载完的内容可能会花费半分钟时间。代码打包文件过大甚至超过 1MB, 当然加载打包文件只是一部分, SPA 还需要向服务器发送 AJAX 请求来加载数据, 此时用户只能耐心地盯着“加载中……”。有些用户因失去耐心而选择离开, 其他人也会非常沮丧。

你希望尽可能快地向用户展示功能性网页, 而不仅仅是一些骨架 HTML 和“加载中……”提示。其他代码可以在用户浏览网页时加载。

使用同构 JavaScript, 可以轻松生成 HTML 来显示服务器上的第一个页面。因此, 当用户加载第一个页面时, 他们将看不到“加载中……”提示。数据在 HTML 中提供给用户使用。他们看到的不再是“加载中……”提示, 而是一个功能性网页, 从而有更好的用户体验。

性能提升主要来自用户不必等待 AJAX 调用。当然也有其他一些优化性能的机会, 比如在 AJAX 调用进入服务器之前预加载数据并将其缓存在服务器上(这正是我们在 DocuSign 通过实施数据路由要做的事情)<sup>3</sup>。

1 Kevin Decker, “Mobile Server Side Rendering,” *GitHub Gist*, 2014, <http://mng.bz/2B6P>

2 Dan Webb, “Improving Performance on twitter.com,” *Twitter*, May 29, 2012, <http://mng.bz/2st9>

3 Ben Buckman, “The New DocuSign Experience, All in Javascript,” *DocuSign Dev*, March 30, 2014, <http://mng.bz/4773>



### 17.1.3 更好的代码可维护性

代码是一种负债。拥有的代码越多，你和你的团队就越需要维护它们。由于这些原因，你希望避免为同一页面设置不同的模板和逻辑。避免重复，不要自我重复(DRY)。

幸运的是，Node.js 是同构 JavaScript 的重要组成部分，因此可以轻松地在服务器上使用前端/浏览器模块。可以在服务器上使用多种模板引擎，比如 Handlebars、Mustache、Dust 等。

鉴于这些问题，我们如何在实际的应用程序中使用同构 JavaScript 来解决这些问题呢？

### 17.1.4 在 React 和 Node 中使用同构 JavaScript

就 Web 开发而言，“同构”意味着在服务器端和客户端都使用相同的代码(通常用 JavaScript 编写)。同构 JavaScript 的一个狭隘用例就是在服务器和客户端上从同一个源渲染。同构 JavaScript 通常意味着使用 JavaScript 和 Node.js，因为这种语言和平台组合允许重用代码库。

浏览器的 JavaScript 代码只需要很少的修改就可以在 Node.js 环境中运行。由于这种互换性，Node.js 和 JavaScript 生态系统具有各种各样的同构框架，如 React.js(<http://facebook.github.io/react>)、Next.js(<https://github.com/zeit/next.js>)、Catberry(<http://catberry.org/>)、LazoJS(<https://github.com/lazojs/lazo>)、Rendr(<https://github.com/rendrjs/rendr>)、Meteor(<https://meteor.com>)等。图 17.2 显示了同构栈的工作原理：同构代码在服务器和客户端之间共享。

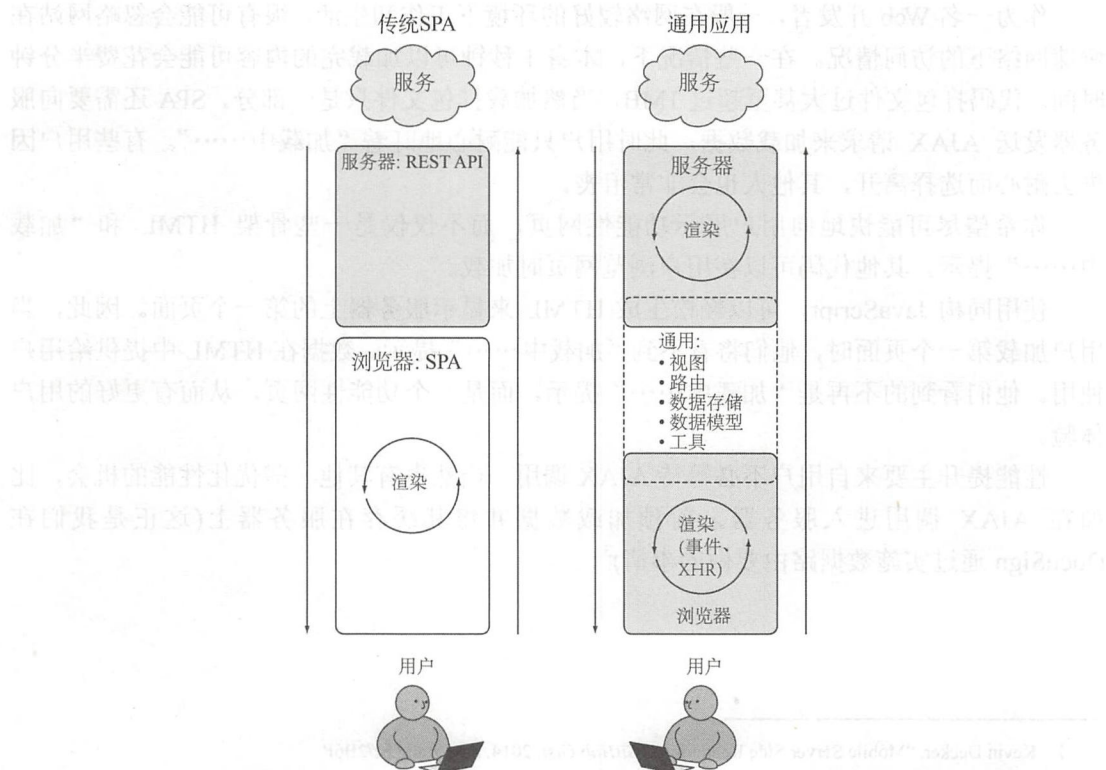


图 17.2 浏览器和服务之间的通用 HTML 生成及代码共享对比传统 SPA 中的无代码共享

在实际应用中，同构 JavaScript 架构由以下部分组成：

- 浏览器的客户端 React 代码。可以用一个 SPA 应用或只是一些简单的用户界面来发送 AJAX 请求。
- 一台 Node.js 服务器，为服务器上的第一页生成 HTML，并将相同的数据提供给浏览器端的 React 代码。可以使用 Express 和模板引擎(或把 React 组件作为模板引擎)来实现。
- Webpack，为服务器和浏览器编译 JSX。

模型如图 17.3 所示。

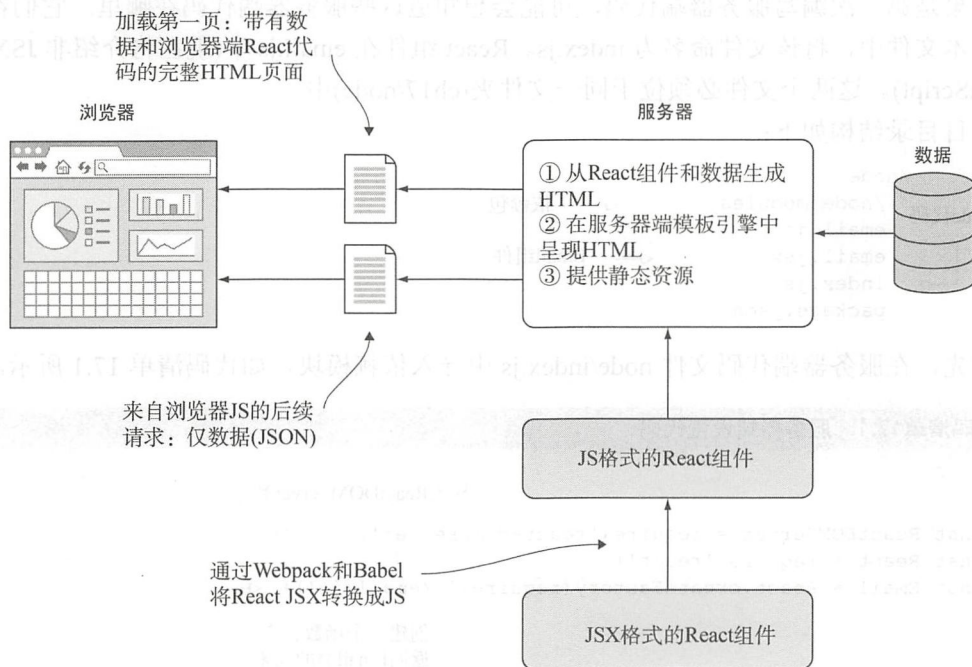


图 17.3 同构 JavaScript 与 React、Node 和 Express 的实际应用

你可能会想，“告诉我如何使用同构 JavaScript 这个奇迹！”。好吧，现在让我们看看在服务器上渲染 React 组件的实例。我们将以渐进的方式去做，因为在使用同构 JavaScript 模式时会涉及多个组件(这里指的不是 React 组件)。你需要学习如何做如下这些事情：

- 从 React 组件生成 HTML：只有 React 组件作为输入，纯 HTML 作为输出；暂无 HTTP(S)服务器。
- 渲染在 Express 服务器中由 React 组件生成的 HTML 代码：类似于前一个项目，但是现在，在模板引擎中使用 React 进行 100%服务器端渲染(没有浏览器 React)。
- 通过 Express 来实现和提供 React 浏览器文件：需要一台 HTTP(S)服务器，Express 只是其中一个选择。到目前为止，你已经使用过 node-static 或 Webpack 开发服务器，但它们并没有在服务器端生成 HTML，只是提供构建/编译好的静态资源。

最后，将使用 React 来生成服务器端的 HTML，同时加载浏览器端的 React——同构 JavaScript 的必杀技。但在你飞跑之前，首先需要学会走路！

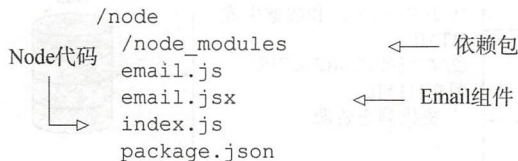
## 17.2 在 Node 上使用 React

让我们从一个基本示例开始：从 Node 脚本生成 HTML。这个例子不包括服务器或任何复杂的东西，只是导入组件和生成 HTML。确保 Node 版本至少是 6，npm 版本至少是 3。

你只需要学习一些在服务器上将 React 组件生成 HTML 的方法。首先，需要 npm 模块 react 和 react-dom。可以按照附录 A 中的说明安装 React 和 npm。本例使用版本 15 的 React 和 React DOM。

如果是第一次编写服务器端代码，可能会想知道这些服务器端代码在哪里。它们在一个纯文本文件中，将该文件命名为 index.js。React 组件在 email.js 中(现在将介绍非 JSX 普通 JavaScript)。这两个文件必须位于同一文件夹(ch17/node)中。

项目目录结构如下：



首先，在服务器端代码文件 node/index.js 中导入依赖模块，如代码清单 17.1 所示。

代码清单 17.1 服务器端设置代码

```
const ReactDOMServer = require('react-dom/server')
const React = require('react')
const Email = React.createFactory(require('./email.js'))
...
// ...
```

Annotations for the code:

- 导入 ReactDOMServer 类 (Import ReactDOMServer class) points to `const ReactDOMServer = require('react-dom/server')`
- 创建一个函数，它返回 Email 类的元素 (Create a function that returns elements of the Email class) points to `const Email = React.createFactory(require('./email.js'))`

什么是 `createFactory()`？如果只是导入 `email.js`，导入结果将是一个组件类；但我们实际需要的是一个 React 元素。因此，需要使用 `createElement()` 或 `createFactory()`。当后者被调用时，会返回一个元素。

导入组件后，运行 `ReactDOMServer.renderToString()`：

```
const emailString = ReactDOMServer.renderToString(Email())
```

下面是 `index.js` 中的代码片段：

```
const ReactDOMServer = require('react-dom/server')
const React = require('react')
const Email = React.createFactory(require('./email.js'))

const emailString = ReactDOMServer.renderToString(Email())
console.log(emailString)
// ...
```



## 导入 JSX

另一种使用 JSX 的方法是即时转换它。babel-register 库可以增强 require，可以 require 它并配置一次，然后就可以像导入任何其他 JS 文件一样导入 JSX。

要导入 JSX，除了安装 babel-register 和 babel-preset-react(使用 npm 来安装它们)之外，还可以使用 babel-register(如 index.js 中所示)：

```
require('babel-register')({
  presets: [ 'react' ]
})
```

email.js 是常规的 JavaScript？在本例中，必须是，参见代码清单 17.2。可以用 Webpack 把 JSX “构建”成 JS。

### 代码清单 17.2 服务器端的 Email 组件(node/email.jsx)

```
const React = require('react')

const Email = (props) => {
  return (
    <div>
      <h1>Thank you {(props.name) ? props.name: '' }
        for signing up!</h1>
      <p>If you have any questions, please contact support</p>
    </div>
  )
}

module.exports = Email
```

你会得到由 React 组件渲染的字符串。可以在你最喜欢的模板引擎中使用这些字符串在网页或其他地方(如 HTML 电子邮件)显示。在本例中，带有标题和段落的 email.js(ch17/node/email.js)使用通用 React 属性渲染为代码清单 17.3 中所示的 HTML 字符串。

### 代码清单 17.3 node/email.jsx 渲染的字符串

```
<div data-reactroot="" data-reactid="1" data-react-checksum="1319067066">
  <h1 data-reactid="2">
    <!-- react-text: 3 -->Thank you <!-- /react-text -->
    <!-- react-text: 4 -->
    <!-- /react-text -->
    <!-- react-text: 5 -->for signing up!<!-- /react-text -->
  </h1>
  <p data-reactid="6">If you have any questions, please contact support</p>
</div>
```

为什么会有属性 data-reactroot、data-reactid 和 data-react-checksum？并没有把它们放在组件中，但 React 却将它们渲染了出来。为什么？因为浏览器端的 React 和同构 JavaScript(在下一节中讨论)。

如果不需要浏览器端 React 所需的 React 标记(例如，如果正在创建 HTML 电子邮件)，请使用 ReactDOMServer.renderToStaticMarkup() 方法。它与 renderToString() 类似，但是去

除了所有的 `data-reactroot`、`data-reactid` 和 `data-react-checksum` 属性。在这种情况下，React 就像其他任何静态模板引擎一样。

例如，可以从 `email.js` 加载组件，并使用 `renderToStaticMarkup()` 而不是 `renderToString()` 生成 HTML：

```
const emailStaticMarkup = ReactDOMServer.renderToStaticMarkup(Email())
```

生成的结果 `emailStaticMarkup` 没有 React 属性：

```
<div><h1>Thank you for signing up!</h1><p>If you have any questions,  
➡ please contact support</p></div>
```

虽然对于 `Email` 不需要使用浏览器端的 React，但是对于 React 的同构 JavaScript 体系架构，需要使用原始的 `renderToString()` 方法。服务器端的 React 以校验和(`data-react-checksum` HTML 属性)的形式向 HTML 中添加了一些秘密。这些校验和由浏览器端的 React 进行比较，如果匹配，那么浏览器组件不需要重新生成/重新绘制/重新渲染，而且不会出现页面闪动(通常由于重新渲染而发生)。如果提供给服务器端组件的数据与浏览器上的数据完全相同，那么校验和将匹配。但是，如何将数据提供给在服务器上创建的组件呢？答案是作为属性提供！

如果需要传递一些属性，请将它们作为对象参数传递。例如，可以为 `Email` 组件提供名称(`Johnny Pineappleseed`)：

```
const emailStringWithName = ReactDOMServer.renderToString(Email({  
  name: 'Johnny Pineappleseed'  
}))
```

接下来显示完整的 `ch17/node/index.js` 代码，有三种方法用来呈现 HTML——`static`、`string` 和带有属性的 `string`：

```
const ReactDOMServer = require('react-dom/server')  
const React = require('react')  
const Email = React.createFactory(require('./email.js'))  
  
const emailString = ReactDOMServer.renderToString(Email())  
const emailStaticMarkup = ReactDOMServer.renderToStaticMarkup(Email())  
console.log(emailString)  
console.log(emailStaticMarkup)  
  
const emailStringWithName =  
➡ ReactDOMServer.renderToString(Email({name: 'Johnny Pineappleseed'}))  
console.log(emailStringWithName)
```

这就是在 Node 中渲染 React 组件到 HTML 的方法——没有服务器，也没有令人激动的代码。接下来，我们来看看如何在 Express 服务器中使用 React。

## 17.3 React 和 Express：在服务器端渲染组件

Express 是最受欢迎的 Node.js 框架之一，并且也许是最流行的。它虽然很简单，但却



高度可配置。有数百个插件被称为中间件，可以和 Express 一起使用这些中间件。

在技术栈的架构图中, Express 和 Node 取代了 HTTP(S)服务器, 有效地替代了 Microsoft IIS([www.iis.net](http://www.iis.net))、Apache httpd(<https://httpd.apache.org>)、nginx([www.nginx.com](http://www.nginx.com))和 Apache Tomcat(<http://tomcat.apache.org>)。Express 和 Node 的独特之处在于它们允许构建高度可扩展的高性能系统, 这要归功于 Node 的非阻塞 I/O 特性(<http://github.com/azat-co/you-dont-know-node>)。Express 的优势在于其广泛的中间件生态系统和成熟稳定的代码库。

遗憾的是, 详细概述这个框架超出了本书的范围, 但是我们将创建一个小型 Express 应用并在其中渲染 React。这绝对不是深入了解 Express, 但它会让你开始使用应用最广泛的 Node.js Web 框架。如果愿意的话, 可以把它叫作快速课程。

提示: 如前所述, 如果想跟进这个示例, 请参考附录 A 来安装 node.js 和 Express。

### 17.3.1 在服务器端渲染简单的文本

使用 Express 构建 HTTP 和 HTTPS 服务器, 然后使用 React 在服务器端生成 HTML, 如图 17.4 所示。在 Express 中使用 React 作为视图引擎的最基本示例是生成一个不带标记(校验和)的 HTML 字符串, 并将其作为响应发送给请求。代码清单 17.4 说明了从 React 组件 about.js 渲染/about 页面的过程。

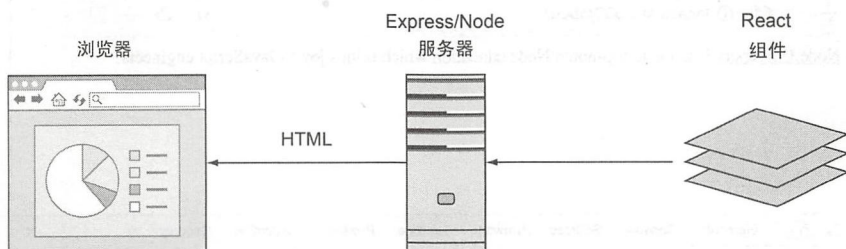


图 17.4 Express/Node 服务器生成 HTML 并将其发送到浏览器

代码清单 17.4 在 Express 上使用 React 在页面上显示 HTML

```
const express = require('express')
const app = express()
const http = require('http')    导入 express 库

const ReactDOMServer = require('react-dom/server')
const React = require('react')  导入 About 组件并创建一个 React 对象
const About =
  React.createFactory(require('./components/about.js'))

app.get('/about', (req, res, next) => {
  const aboutHTML = ReactDOMServer.renderToStaticMarkup(About())
  response.send(aboutHTML)
})

http.createServer(app)           实例化 HTTP 服务器并启动
  .listen(3000)

  响应 /about 请求, 将 HTML 字符串发送回客户端
```

以上代码可以正常工作, 但/about 不是一个含有<head>和<body>的完整页面。最好使



用适当的模板引擎(如 Handlebars)作为布局和顶部的 HTML 元素。你也许想知道 `app.get()` 和 `app.listen()` 是什么。我们来看另一个示例, 通过这个示例就能明白这些问题了。

### 17.3.2 渲染 HTML 页面

这是一个更有趣的例子, 在本例中将使用一些外部插件和一个模板引擎。应用的功能是相同的: 服务器端使用 Express 从 React 生成 HTML。页面将显示一些由 `about.jsx` 生成的文本(见图 17.5)。没有令人激动的代码, 而且很简单, 但从简单开始是不错的选择。

创建名为 `react-express` 的文件夹。(这个示例在 `ch17/react-express` 中)最终的项目结构如下:

```
/react-express
  /components
    about.jsx
  /views
    about.hbs
    index.js
  package.json
```

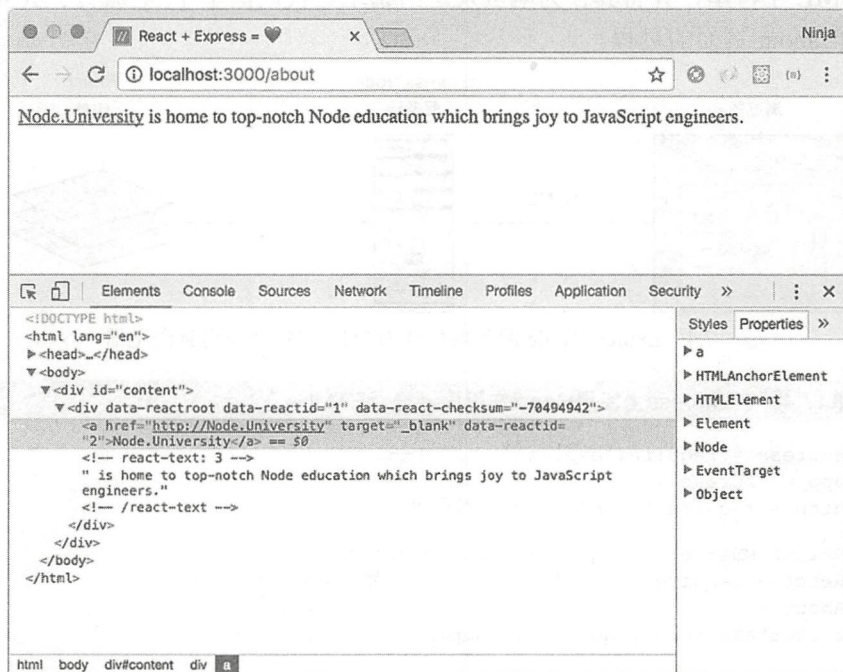


图 17.5 从服务器上的 React 组件进行渲染

使用 `npm init -y` 创建 `package.json`, 然后像下面这样使用 `npm` 安装 Express:

```
$ npm install express@4.14.0 --save
```

与任何 Node 应用一样, 打开一个编辑器并创建一个文件。通常创建一个名为 `index.js`、`app.js` 或 `server.js` 的服务器文件, 稍后使用 `node` 命令运行该文件。在本例中, 将其命名为

index.js。

该文件包含如下部分：

- 导入：导入需要的依赖，例如 `express` 及其插件。
- 配置：设置某些配置项，例如要使用的模板引擎。
- 中间件：定义对所有传入请求执行的常见操作，如验证、认证、压缩等。
- 路由：定义服务器处理的 URL，例如 `/accounts/`、`/users` 等，以及对它们的操作。
- 错误处理程序：发生错误时显示有意义的消息或网页。
- 启动服务：启动 HTTP 或 HTTPS 服务器。

以下是对 Express 和 Node 服务器文件的详细概述：

```
const express = require('express')           ◀—— 导入模块
const app = express()
const errorHandler = require('errorhandler')
const http = require('http')
const https = require('https')
// Import other modules
// ...

app.set('view engine', 'hbs')                ◀—— 设置配置

app.get('/',
  // ...
)
app.get('/about',
  // ...
)
// ...

app.use(errorHandler)                        ◀—— 定义错误处理程序
                                           (中间件的类型)

http.createServer(app)                       ◀—— 启动HTTP服务器
  .listen(3000)

// ...
if (typeof options !== 'undefined')
  https.createServer(app, options)           ◀—— 启动HTTPS服务器
  .listen(443)
```

现在我们来深入一些。导入部分很简单，在代码中导入依赖并实例化对象。例如，导入 Express 框架并创建实例，代码如下：

```
var express = require('express')
var app = express()
```

### 配置

可以使用 `app.set()` 来设置配置，其中第一个参数是一个字符串，第二个参数是一个值。例如，使用如下代码配置 `view engine` (视图引擎) 将模板引擎设置为 `hbs` (`www.npmjs.com/package/hbs`)：

```
app.set('view engine', 'hbs')
```

hbs 是 Handlebars 模板语言(<http://handlebarsjs.com>)的 Express 模板(或视图)引擎。你可能已经使用过 Handlebars 或类似的引擎,如 Mustache、Blaze 等。Ember 也使用 Handlebars(<http://mng.bz/90Q2>)。它是一个常用的、易于启动的模板,这就是为什么会在这里使用它的原因。

**警告:** 必须使用 `npm i hbs --save` 来安装 hbs 包,以便 Express 能正确使用视图引擎。

### 中间件

下一节将设置中间件。例如,要使应用能够提供静态资源,请使用 static 中间件:

```
app.use(express.static(path.join(__dirname, 'public')))
```

static 中间件非常棒,因为它将 Express 转换为一台静态 HTTP(S)服务器,从而代理请求到指定文件夹中的文件(在本例中为 public),就像 NGINX 或 Apache httpd 一样。

### 路由

接下来是路由,也称为端点、资源、页面和许多其他名称。可以定义一个 URL 模式,该模式通过 Express 与传入请求的实际 URL 进行匹配。如果匹配成功,Express 将执行与此 URL 模式相关的逻辑,这种方式被称为处理请求。它可能涉及任何内容,从为 404 Not Found 页面显示静态 HTML 到发送请求到另一个服务,并在响应发送回客户端之前缓存响应。

路由是 Web 应用中最重要的一部分,因为它们定义了 URL 路由,并以一种方式充当传统 MVC(模型-视图-控制器)模式中的控制器。在 Express 中,使用 `app.NAME()` 模式定义路由,其中 NAME 是小写的 HTTP 方法名。例如,如下代码是 GET /(首页或空 URL)路由的语法,该路由将返回字符串“Hello”:

```
app.get('/', (request, response, next) => {  
  response.send('Hello!')  
})
```

对于/about 页面/路由,可以更改第一个参数(URL 模式)。响应也可以返回 HTML 字符串:

```
app.get('/about', (req, res, next) => {  
  response.send(`

<a href="https://node.university" target="_blank">Node.University</a>  
    is home to topnotch  
    Node education which brings joy to JavaScript engineers.  
  </div>`)  
})


```

### 使用 Handlebars 布局

接下来,要从 Handlebars 模板渲染 React HTML,因为 Handlebars 会提供包含诸如 `<html>` 和 `<body>` 之类的整体布局。换句话说,页面既有 React 的 UI 元素,也有 Handlebars 的布局。

创建一个包含 about.hbs 模板视图的文件夹,其中 about.hbs 里的代码如下:



```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>React + Express = ❤️ </title>  ← 使用 ❤️
    <meta name="author" content="Azat" />
  </head>

  <body>
    <div id="content">{{{about}}}</div>  ← 使用三重花括号从about变量输出未转义的HTML(在index.js中提供)
  </body>
</html>

```

### 渲染页面

在路由中(在文件 `ch17/react-express/index.js` 中), 将 `response.send()` 更改为 `response.render()`:

```

// ...
const React = require('react')
require('babel-register')({
  presets: [ 'react' ]
})
const About =
  React.createFactory(require('../components/about.jsx'))
// ...
app.get('/about', (request, response, next) => {
  const aboutHTML = ReactDOMServer.renderToString(About())
  response.render('about', {about: aboutHTML})
})
// ...

```

增强实时转换JSX功能, 使 import/require 可以直接导入 JSX 文件

准备 About 组件

将 React HTML 字符串传递给 Handlebars 模板 `about.hbs`

使用 React 标记生成 React HTML 字符串

Express 路由可以从 Handlebars 模板中渲染具有诸如 `about` 字符串变量之类的数据, 或者以字符串格式发送响应。

### 是否必须使用不同的模板引擎来进行服务器渲染和布局?

使用 React 替代 Handlebars 进行布局是可行的。有一个 `express-react-views` 库可以做到这一点([www.npmjs.com/package/express-react-views](http://www.npmjs.com/package/express-react-views))。这个库只能用来渲染静态标记, 而不能用于浏览器端的 React。

我不会在这里讲解它, 因为它需要使用 `dangerouslySetInnerHTML`<sup>4</sup>, 而且不支持所有的 HTML, 并且经常会使初学 Express-React 的开发者感到迷惑。在我看来, 使用 React 进行布局几乎没有什么好处。

### 错误处理

错误处理程序类似于中间件。例如, 可以导入 `errorhandler` 包([www.npmjs.org/package/errorhandler](http://www.npmjs.org/package/errorhandler)):

```

const errorHandler = require('errorhandler')
...

```

4 请参阅第 3 章或 <https://facebook.github.io/react/docs/dom-elements.html#dangerouslysetinnerhtml>

```
app.use(errorHandler)
```

或者在 `index.js` 中创建它们:

```
app.use((error, request, response, next) => {
  console.error(request.url, error)
  response.send('Wonderful, something went wrong...')
})
```

通过调用请求处理程序或中间件中的 `next(error)` 来触发错误处理程序。`error` 是一个错误对象, 可以使用 `new Error('Ooops')` 创建错误对象, 其中“Ooops”为错误消息。这是 `/about` 中的一个示例:

```
app.get('/about', (request, response, next) => {
  // ... do weird stuff
  let somethingWeirdHappened = true
  if (somethingWeirdHappened) return next(new Error('Ooops'))
})
```

不要忘记使用 `return`。有关 Node 和 Express 中错误处理的更多信息, 请查看 Node Patterns 课程(<http://node.university/p/node-patterns>)或我的文章“Node Patterns: From Callbacks to Observer”(<http://webapplog.com/node-patterns>)。

### 启动服务器

最后, 通过传递一个端口号和一个回调(可选)给 `listen()` 并执行来启动应用:

```
http.createServer(app).listen(portNumber, callback)
```

在本例中, 如下所示:

```
http.createServer(app)
  .listen(3000)
```

代码清单 17.5 是 `ch17/react-express/index.js` 的完整服务器代码, 确保没有遗漏任何代码。

#### 代码清单17.5 React、Express、hbs服务器的完整代码

```
const fs = require('fs')
const express = require('express')
const app = express()
const errorHandler = require('errorhandler')

const http = require('http')
const https = require('https')

const React = require('react')
require('babel-register')({
  presets: [ 'react' ]
})
```

```

const ReactDOMServer = require('react-dom/server')
const About = React.createFactory(require('./components/about.jsx'))

app.set('view engine', 'hbs')
app.get('/', (request, response, next) => {
  response.send('Hello!')
})

app.get('/about', (request, response, next) => {
  const aboutHTML = ReactDOMServer.renderToString(About())
  response.render('about', {about: aboutHTML})
})

app.all('*', (request, response, next) => {
  response.status(404).send('Not found...
  ➡ did you mean to go to /about instead?')
})
app.use((error, request, response, next) => {
  console.error(request.url, error)
  response.send('Wonderful, something went wrong...')
})

app.use(errorHandler)

http.createServer(app)
  .listen(3000)

try {
  const options = {
    key: fs.readFileSync('./server.key'),
    cert: fs.readFileSync('./server.crt')
  }
} catch (e) {
  console.warn('Create server.key and server.crt for HTTPS')
}
if (typeof options !== 'undefined')
  https.createServer(app, options)
    .listen(443)

```

实现捕获所有的URL。你无法相信有那么多人 在我的课堂上实现一台 服务器，然后请求一个 不存在的网址，并认为 代码出现了错误，实际 上他们应该访问/about

加载SSL/HTTPS 的密钥和证书<sup>5</sup>

现在使用 `node index.js` 或快捷方式(`node.`)运行服务器，然后使用浏览器打开 `http://localhost:3000/about` 以查看服务器响应。如果在启动服务器并打开地址时发生错误或者出现错误，请参阅 `ch17/react-express` 中的项目源代码。

**警告：**SSL 和 HTTPS 需要 SSL 密钥和证书才可以正常工作。本例的 GitHub 代码故意不包含 `server.key` 和 `server.crt`，因为密钥等敏感信息不应该提交给版本控制系统。应该按照 <https://webapplog.com/http2-node> 上的说明创建自己的密钥。如果没有密钥，那么示例代码将只创建一台 HTTP 服务器。

最终的结果应该是一个包含标题和正文的 HTML 页面。在正文中应该有 React 标记，如 `data-react-checksum` 和 `data-reactroot`，如图 17.6 所示。

<sup>5</sup> 要了解如何生成密钥和证书，可查我的博客贴子“Easy HTTP/2 Server with Node.js and Express.js”，<https://webapplog.com/http2-node>



为什么这个例子使用标记而不是静态的 HTML 字符串或 `express-react-views` 渲染？因为稍后同构 JavaScript 架构的浏览器端 React 将需要使用带校验和的标记。

在下一节中，将通过把整合所有相关知识(浏览器端 React、Express 和 Node 端 React)来实现同构 JavaScript 架构。

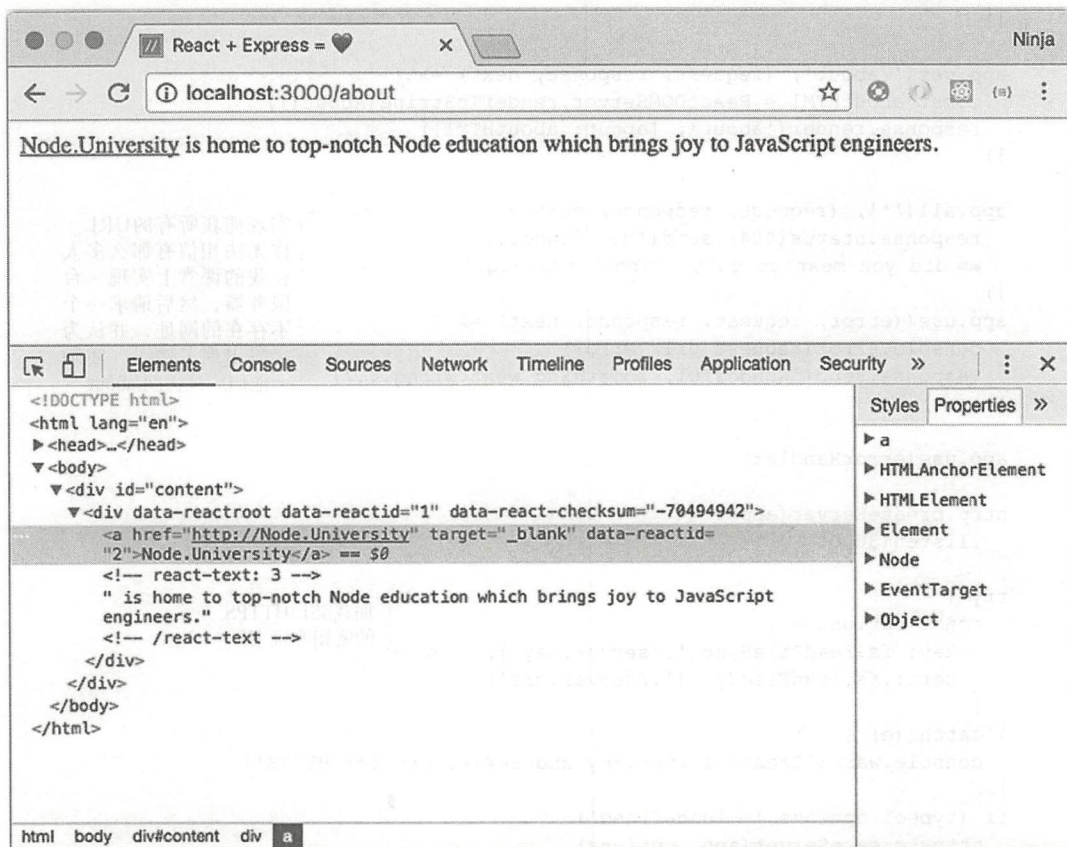


图 17.6 使用 Express 来渲染来自 Handlebars 布局的 React 标记

## 17.4 使用 Express 和 React 的同构 JavaScript

本节将结合本章(以及本书大部分内容)中的所有技能。你将在服务器上渲染组件，将它们插入到模板中，并启用浏览器端的 React。

构建一个包含如下三个组件的留言板：Header、Footer 和 MessageBoard(见图 17.7)，用来学习同构 JavaScript。Header 和 Footer 组件由静态的 HTML 来显示一些文本，MessageBoard 组件具有用于在留言板上发布消息的表单和消息列表。这个应用使用 AJAX 调用来获取消息列表并将新消息发送到后端服务器，而后端服务器使用 MongoDB NoSQL 数据库。

简而言之，对于通用 React，需要遵循以下步骤：

- ① 设置服务器，以便向模板提供数据并渲染 HTML(组件和属性)，如 `index.js`。
- ② 创建输出数据的模板，如 `views/index.hbs`。

③ 在模板中包含用于交互的浏览器端 React 文件(ReactDOM.Render), 例如 client/app.jsx。

④ 创建 Header、Footer 和 MessageBoard 组件。

⑤ 使用 Webpack 设置构建过程, 例如 webpack.config.js。

以下几个部分相互作用: 服务器、组件、数据和浏览器。图 17.8 显示了它们在留言板示例中的连接方式。服务器充当静态资源 HTTP 服务器, 并作为渲染服务器端 HTML 的应用(仅限第一次加载页面)。浏览器端 React 代码在初始页面加载之后启用浏览器交互事件和后续的持久性操作(借助对服务器的 HTTP 请求)。

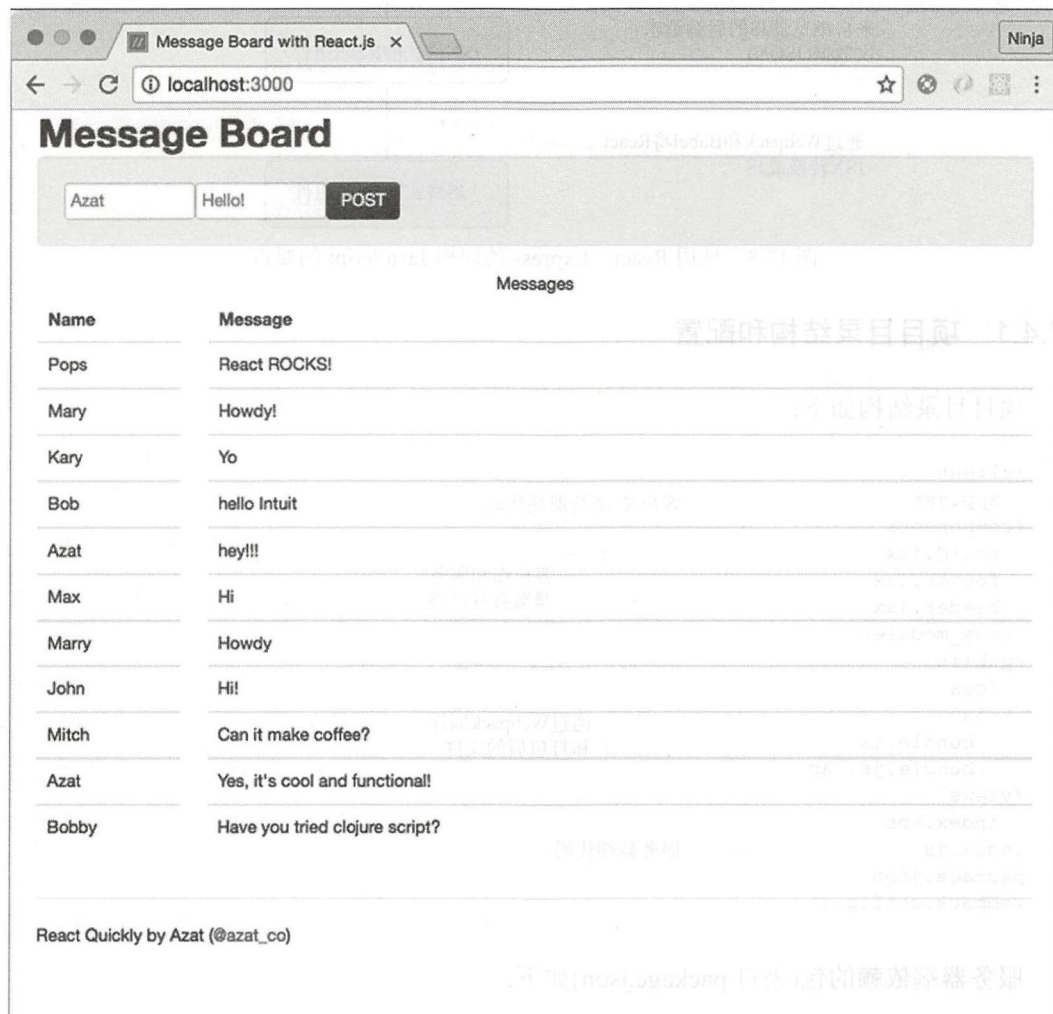


图 17.7 包含发布消息表单和消息列表的留言板应用

注意: 需要安装并启动 MongoDB 才能使此例正常工作。可以在网站或附录 D 中查看安装方法。安装 MongoDB 后, 运行 mongod 并保持运行。Express 服务器使用神奇的 URL mongodb://localhost:27017/board 连接到 MongoDB。

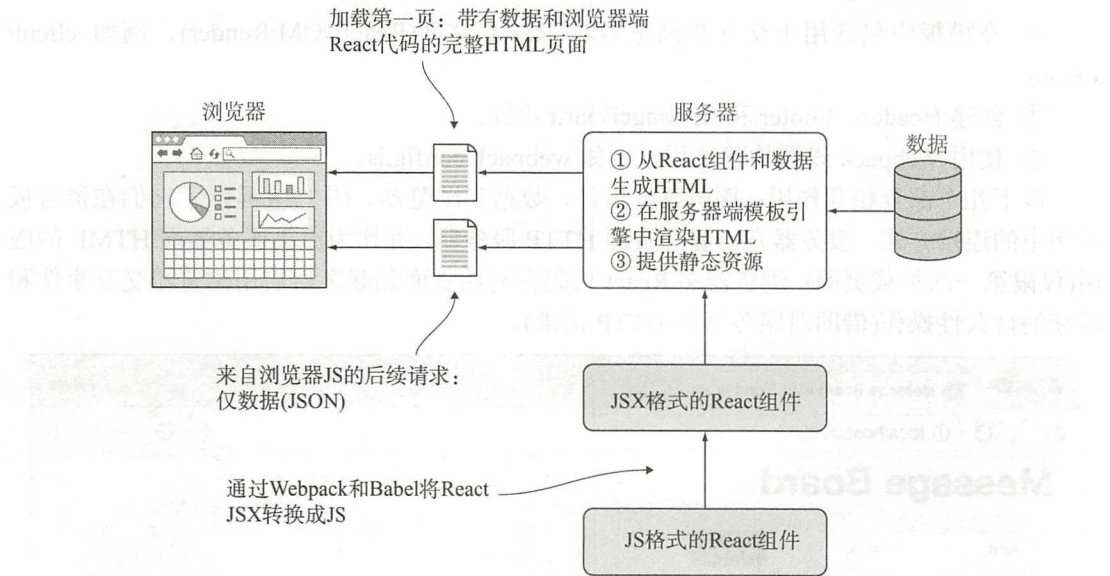


图 17.8 使用 React、Express 的同构 JavaScript 的要点

### 17.4.1 项目目录结构和配置

项目目录结构如下：

```

/client
  app.jsx
/components
  board.jsx
  footer.jsx
  header.jsx
/node_modules
/public
  /css
  /js
    bundle.js
    bundle.js.map
/views
  index.hbs
index.js
package.json
webpack.config.js

```

← 客户端/浏览器端代码

← 客户端和服务端共有代码

← 通过 Webpack 编译和打包后的文件

← 服务器端代码

服务器端依赖的包(来自 package.json)如下：

```

...
"dependencies": {
  "babel-register": "6.11.6",
  "body-parser": "1.13.2",
  "compression": "1.5.1",
  "errorhandler": "1.4.1",
  "express": "4.13.1",
  "hbs": "4.0.0",

```

← 在 Node 中使用 require 加载 JSX

← 使用 Express 框架



在服务器端使用 React 渲染

```
"express-validator": "2.13.0",
"mongodb": "2.2.6",
"morgan": "1.6.1",
"react": "15.5.4",
"react-dom": "15.5.4"
},
...
```

使用 MongoDB 来存储消息(这是驱动程序,在实际应用中还需要数据库)

现在开始在 `message-board/index.js` 中设置服务器。

### Express 中间件

我想就这个项目中使用的中间件讲几句话,刚接触 Express 的读者不妨听一听。Express 不是一个做所有事情的大框架。相反,它是一个基础层,在这个基础层之上,Node 工程师构建了他们自己的框架,使它们恰好适合自己手头的任务,而一体化框架并不能总是适合自己的任务。你只需要使用 Express 及其插件生态系统即可获得所需的内容。这些插件称为中间件,因为它们使用中间件模式,并使用 Express 实现中间件管理器。

每个 Express 工程师都有自己在项目中喜欢使用的中间件包。我个人倾向于从以下包开始,然后在需要时添加更多的包:

- `compression`: 使用 `gzip` 算法自动压缩响应。这使得下载的响应更小、更快。
- `errorhandle`: 例如 404 和 500 错误的基本处理程序。
- `express-validator`: 验证传入请求的有效参数。
- `morgan`: 记录服务器上的请求,支持多种格式。
- `body-parser`: 启用 JSON 和 `urlencoded` 数据格式的自动解析,从而在 `request.body` 中访问 Node/JS 对象。

更多有关 `compression`、`body-parser` 和 `errorhandler` 以及其他 Express 中间件的信息,请参阅附录 C、网址 <https://github.com/azatco/cheatsheets/tree/master/express4> 或图书 *Pro Express.js* (<http://proexpressjs.com>)。

## 17.4.2 启动服务器

就像在前面示例中所做的那样,在 `index.js` 中实现服务器端的功能,然后通过五个部分来看看它是如何被分解的。首先,代码清单 17.6 中是完整的代码(`ch17/message-board/index.js`)。

代码清单 17.6 留言板应用的服务器端代码

```
require('babel-register')({
  presets: [ 'react' ]
})

const express = require('express'),
      mongodb = require('mongodb'),
      app = express(),
      bodyParser = require('body-parser'),
      validator = require('express-validator'),
      logger = require('morgan'),
      errorHandler = require('errorhandler'),
      compression = require('compression'),
      url = 'mongodb://localhost:27017/board',
```

← 为导入的 JSX 定义显示的名称  
并将其即时编译为 JS HOC

← 定义本地 MongoDB 实例的地址  
以及数据库名称(board)

```

ReactDOMServer = require('react-dom/server'),
React = require('react')

const Header = React.createFactory(require('./components/header.jsx')),
  Footer = React.createFactory(require('./components/footer.jsx')),
  MessageBoard = React.createFactory(require('./components/board.jsx'))

mongodb.MongoClient.connect(url, (err, db) => {
  if (err) {
    console.error(err)
    process.exit(1)
  }

  app.set('view engine', 'hbs')

  app.use(compression())
  app.use(logger('dev'))
  app.use(errorHandler())
  app.use(bodyParser.urlencoded({extended: true}))
  app.use(bodyParser.json())
  app.use(validator())
  app.use(express.static('public'))

  app.set('view engine', 'hbs')

  app.use((req, res, next) => {
    req.messages = db.collection('messages')
    return next()
  })

  app.get('/messages', (req, res, next) => {
    // ...
  })
  app.post('/messages', (req, res, next) => {
    // ...
  })

  app.get('/', (req, res, next) => {
    // ...
  })

  app.listen(3000)
})

```

使用URI连接到 MongoDB实例

将集合设置为请求对象的属性，以便在其他路由及其模块化中访问

### 配置

再次提醒，安装 `babel-register` 和 `babel-preset-react` 之后，需要使用 `babel-register` 来导入 JSX：

```

require('babel-register')({
  presets: [ 'react' ]
})

```

在 `index.js` 中，实现了 Express 服务器。我们使用相对路径 `./components/` 来导入组件：

```

const Header = React.createFactory(require('./components/header.jsx')),
  Footer = React.createFactory(require('./components/footer.jsx')),
  MessageBoard = React.createFactory(require('./components/board.jsx'))

```



为了渲染 React 应用, 需要知道 Express 几乎可以使用任何模板引擎。在本例中我们考虑使用接近常规 HTML 的 Handlebars 模板引擎。如果 app 是 Express 实例, 那么可以使用如下语句启用 Handlebars:

```
app.set('view engine', 'hbs')
```

必须安装 hbs 模块(在 package.json 中已经包含该模块)。

### 中间件

中间件为服务器提供了许多功能, 否则就必须自己实现它们。以下是这个项目最基本的部分:

```
// ...
app.use(compression())
app.use(logger('dev'))
app.use(errorHandler())
app.use(bodyParser.urlencoded({extended: true}))
app.use(bodyParser.json())
app.use(validator())
app.use(express.static('public'))
// ...
```

启用对所传入 JSON 数据的解析 →

启用请求的服务器日志, 用来帮助进行调试和开发 ←

允许访问 public 下的所有文件, 比如 bundle.js ←

### 服务器端路由

在路由/中, 通过调用 render() 方法来渲染 views/index.handlebars(res.render("index")) 模板, 之所以是 views 文件夹, 是因为默认的模板文件夹是 views:

```
app.get('/', (req, res, next) => {
  req.messages.find({}, {sort: {_id: -1}}).toArray((err, docs) => {
    if (err) return next(err)
    res.render('index', data)
  })
})
```

req.message.find() 是 MongoDB 获取文档的方法。虽然必须安装并运行 MongoDB 才能正常运行(没有任何改变), 但是我不喜欢强迫你使用我选择的数据库。可以很容易地用任何想要的东西来代替对 MongoDB 的调用。大多数现代的 RDBMS 和 NoSQL 数据库都有 Node 驱动程序; 它们中的大多数甚至有使用 Node 编写的 ORM/ODM 库。因此, 如果不打算使用 MongoDB, 则可以放心地忽略数据库调用。如果想使用 MongoDB, 可以查看附录 D 中的速查表。上述主要思想是: 在请求处理程序中, 可以调用外部服务(例如, 使用 axios 获取 Facebook 用户信息)或使用数据库(MongoDB、PostgreSQL 等)。但是如何获取 Node 中的数据不是本章讨论的重点。

这里关于通用 React 最重要的是 res.render() (ch17/message-board/index.js), 如代码清单 17.7 所示。render() 方法是 Express 用于模板的一个特性。该方法有两个参数。第一个参数是模板的名称 index.hbs, 位于 views 目录中。第二个参数是 res.locals, 指定在模板中使用的数据。所有的数据被发送(结合或融合)到 ch17/message-board/views/index.hbs 模板(.hbs 扩展名是可选的)。



代码清单17.7 渲染从React组件生成的HTML

```

...
app.set('view engine', 'hbs')
...
    从MongoDB请求一组根据ObjectID逆向排序的消息
app.get('/', (req, res, next) => {
  req.messages.find({},
    {sort: {_id: -1}}).toArray((err, docs) => {
    if (err) return next(err)
    res.render('index', {
      header: ReactDOMServer.renderToString(Header()),
      footer: ReactDOMServer.renderToString(Footer()),
      messageBoard: ReactDOMServer.renderToString(MessageBoard({
        messages: docs
      })),
      props: '<script type="text/javascript">var messages='
        +JSON.stringify(docs)
        +'</script>'
    })
  })
})

```

应用Handlebars模板引擎

发送从Header生成的HTML字符串

发送从Footer生成的HTML字符串

向浏览器端React发送消息列表

发送一个HTML字符串，它由将消息列表(docs)作为属性的MessageBoard组件生成

此时，你拥有一台 Express 服务器，该服务器使用来自 React 组件的三个 HTML 字符串渲染 Handlebars 模板。这本身并不令人兴奋，即便不使用 React 也可以做到这一点。可以使用 Handlebars、Pug、Mustache 或任何其他模板引擎来渲染一切，而不仅仅是布局。但为什么还需要 React？因为要在浏览器上使用 React，而浏览器端 React 将获取服务器的 HTML 并添加所有的事件和状态。

到目前为止，还没有完成服务器。还需要为这个示例实现两个 API：

- GET /messages: 获取数据库中的消息列表。
- POST /messages: 在数据库中创建一条新的消息。

这些路由在浏览器端 React 使用 AJAX/XHR 请求 GET 和 POST 数据时使用。Express 路由的代码在 index.js 中：

```

app.get('/messages', (req, res, next) => {
  req.messages.find({},
    {sort: {_id: -1}}).toArray((err, docs) => {
    if (err) return next(err)
    return res.json(docs)
  })
})

```

处理创建消息(POST /message)的路由使用 express-validator 来确保传入的数据存在 (notEmpty())。express-validator 是一个非常方便的中间件，可以用它来设置各种验证规则。

**警告：**输入验证对于保护应用至关重要。开发者使用代码和系统：他们编写代码，了解代码的工作原理，并知道它们支持哪些数据。因此，他们会无意识地给应用提供片面的数据，这可能会导致系统漏洞。所以你要始终在服务器端对数据进行无害化处理。你应该将每个用户都视为潜在的恶意攻击者或疏忽大意的人，他们从不阅读你的说明，并且总是

发送奇怪的数据。

下面这个路由使用来自 req.messages 的数据库引用插入新的消息：

```
app.post('/messages', (req, res, next) => {
  req.checkBody('message',
    'Invalid message in body').notEmpty()
  req.checkBody('name', 'Invalid name in body').notEmpty()
  var errors = req.validationErrors()
  if (errors) return next(errors)
  req.messages.insert(req.body, (err, result) => {
    if (err) return next(err)
    return res.json(result.ops[0])
  })
})
```

检查消息是否存在于请求体中

将请求体插入到数据库中

输出由数据库自动生成的新文档的ID

### node-dev

如前所述，我推荐使用 nodemon 或类似的工具，比如 node-dev。node-dev 监视文件更改并在检测到更改时重新启动服务器。它可以节省你的工作时间！要安装 node-dev，请运行以下命令：

```
npm i node-dev@3.1.3 --save-dev
```

在 package.json 中，可以添加命令 node-dev 到 npm 脚本 start 中：

```
...
"scripts": {
  ...
  "start": "./node_modules/.bin/webpack && node-dev ."
},
...
```

与前一节使用 HTTPS 的部分相比，启动服务器的调用是比较原始的：

```
app.listen(3000)
```

当然，也可以添加 HTTPS 并更改端口号或从环境变量中获取端口号。

请记住，在本例中，根路由/会处理所有到/或到 http://localhost:3000/的 GET 请求。它的实现在代码清单 17.7(ch17/message-board/index.js)中。该路由在 res.render()中使用名为 index 的模板。现在，我们来实现这个模板。

### 17.4.3 使用 Handlebars 的服务器端布局模板

可以在服务器上使用任何模板引擎来渲染 React HTML。和 HTML 类似的 Handlebars 是不错的选择，从 HTML 转换到这个模板引擎时只需要做很少的修改。以下是 Handlebars index.hbs 文件：

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <!-- meta tags and CSS -->
```



```

</head>

<body>
<div class="container-fluid">
  <!-- header -->
  <!-- props -->
  <!-- messageBoard -->
  <!-- footer -->
</div>
<script type="text/javascript" src="/js/bundle.js"></script>
</body>
</html>

```

使用三个花括号({{...}})来输出组件和属性(非转义输出),例如 HTML。{{props}}将输出一个<script/>脚本标签,以便可以在脚本中定义 message 变量。用于渲染非转义 HTML 字符串 props 的 index.hbs 代码是<div>{{props}}</div>。

其余的数据也都采用类似的输出方式:

```

<div id="header">{{header}}</div>
...
<div>{{props}}</div>
...
<div class="row-fluid" id="message-board" />{{messageBoard}}</div>
...
<div id="footer">{{footer}}</div>

```

代码清单 17.8 中是如何在 Handlebars 模板中将 Header 组件输出为 HTML 字符串的代码(ch17/message-board/views/index.hbs)。

代码清单17.8 在Handlebars中输出由React生成的HTML

```

...
<div class="container-fluid">
  <div class="row-fluid">
    <div class="span12">
      <div id="header">{{header}}</div>
    </div>
  </div>
  ...

```

数据呢?为了使服务器端 React 与浏览器端 React 一起工作,必须在浏览器和服务端上使用相同的数据创建 React 元素。可以通过将数据作为 JS 变量嵌入到 HTML 中,把数据从服务器传递到浏览器端 React,而不用通过调用 AJAX 来获取数据。

当传递 header、footer 和 messageBoard 时,可以在 Express 路由/中添加 props。在模板文件 index.hbs(ch17/message-board/views/index.hbs)中,用三个花括号将值打印出来,该文件还包含稍后由 Webpack 打包生成的 js/bundle.js 脚本,参见代码清单 17.9。



代码清单 17.9 从 React 组件渲染 HTML 的服务器端布局

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Message Board with React.js</title>
    <meta name="description" content="Message Board" />
    <meta name="author" content="Azat Mardan" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link type="text/css" rel="stylesheet" href="/css/bootstrap.min.css" />
    <link type="text/css" rel="stylesheet"
      href="/css/bootstrap-responsive.min.css" />
  </head>
  <body>
    <div class="container-fluid">
      <div class="row-fluid">
        <div class="span12">
          <div id="header">{{{header}}}</div>
        </div>
      </div>
      <div>{{{props}}}</div>
      <div class="row-fluid">
        <div class="span12">
          <div id="content">
            <div class="row-fluid" id="message-board" />{{{messageBoard}}}</div>
          </div>
        </div>
        <div class="row-fluid">
          <div class="span12">
            <div id="footer">{{{footer}}}</div>
          </div>
        </div>
      </div>
      <script type="text/javascript" src="/js/bundle.js"></script>
    </body>
  </html>

```

输出从 Header 组件生成的 HTML

输出包含 <script> 脚本的 HTML，并且脚本中包含一个消息列表的数组

引入浏览器端所需的 React 代码

该模板包含一些 Twitter Bootstrap 样式，但对于该项目或同构 JavaScript 示例来说并不重要。在模板中使用了一些变量(也就是 `res.locals: header、messageBoard、props` 和 `footer`)，这些变量需要在 Express 请求处理程序的 `render()` 中提供。提醒一下，下面是之前实现的 `index.js` 代码(参见代码清单 17.7，`ch17/message-board/index.js`)，该代码通过调用 `index`(按约定是指 `index.hbs`)来使用之前的模板：

```

res.render('index', {
  header: ReactDOMServer.renderToString(Header()),
  footer: ReactDOMServer.renderToString(Footer()),
  messageBoard:
    ReactDOMServer.renderToString(MessageBoard({messages: docs})),
  props: '<script type="text/javascript">var messages='+JSON.stringify(docs)+
    '</script>'
})

```

这些变量值由 React 组件生成。通过这种方式，可以在服务器和浏览器上使用相同的组件。在服务器上(使用 Node)轻松渲染的能力是 React 的美妙之处。

接下来，我们继续讨论变量 props、header、footer 等。

#### 17.4.4 在服务器上编写 React 组件

最后，需要做前几章所做的事情：创建 React 组件。偶尔回到一些熟悉的东西不是很好吗？是的。但组件从哪里来？它们被放在组件文件夹中。正如前面提到的，组件在浏览器和服务器的使用，这就是为什么把它们放在单独的组件文件夹中，而不是在客户端创建它们的原因(组件文件夹的名称应该是共享且通用的)。

为了暴露这些组件，每一个组件都必须具有 `module.exports`，并将一个组件类的值或一个无状态函数赋值给 `module.exports`。例如，引入 React，实现类或函数，然后按如下所示导出 Header：

```
const React = require('react')
const Header = () => {
  return (
    <h1>Message Board</h1>
  )
}
module.exports = Header
```

导出这个无状态组件

声明一个无状态组件

虽然在代码中没有提及 React，但 JSX 需要使用它

留言板使用 AJAX/XHR 调用来获取消息列表并发布新消息。这些调用在 `board.jsx` 文件中进行，该文件包含 `MessageBoard` 组件，这个组件是你的容器(智能)组件，所以这些 AJAX/XHR 调用都在这个组件中进行。

在 `MessageBoard` 中进行 AJAX 调用的地方很有趣：在 `componentDidMount()` 中，因为这个生命周期事件永远不会在服务器上调用，参见代码清单 17.10(ch17/message-board/components/board.jsx)。

代码清单 17.10 获取消息列表和发布新消息

```
const request = require('axios')
const url = 'http://localhost:3000/messages'
const fd = ReactDOM.findDOMNode
...
class MessageBoard extends React.Component {
  constructor(ops) {
    super(ops)
    this.addMessage = this.addMessage.bind(this)
    if (this.props.messages)
      this.state = {messages: this.props.messages}
  }
  componentDidMount() {
    request.get(url, (result) => {
      if(!result || !result.length){
        return;
      }
      this.setState({messages: result})
    })
  }
  addMessage(message) {
    let messages = this.state.messages
```

为服务器地址创建一个变量，可以稍后改变它

使用 axios 进行 GET 请求，成功后使用消息列表更新状态

```

    request.post(url, message)
      .then(result => result.data)
      .then((data) => {
        if (!data) {
          return console.error('Failed to save')
        }
        console.log('Saved!')
        messages.unshift(data)
        this.setState({messages: messages})
      })
  }
}

render() {
  return (
    <div>
      <NewMessage messages={this.state.messages} addMessageCb=
        {this.addMessage} />
      <MessageList messages={this.state.messages} />
    </div>
  )
}

```

使用 axios 进行 POST 请求，成功后通过更新状态将消息添加到消息列表中

传递添加消息的方法到 NewMessage 展示/木偶组件中，它负责创建表单和事件侦听器

可以在同一个文件(ch17/message-board/components/board.jsx)中查看 NewMessage 和 MessageList 的实现，此处不再赘述。这两个组件是具有很少逻辑或无逻辑的展示组件——只是以 JSX 的形式描述了 UI。

至此，已经完成了在服务器上渲染 React(和布局)HTML。现在，让我们使用浏览器端的 React 同步标记，否则，将无法添加任何消息，因为没有交互式浏览器的 JavaScript 事件。

### 17.4.5 客户端 React 代码

如果就此停止继续开发，那么服务器上渲染的 React 组件就只是静态标记而已。新消息不会被保存，因为 POST 按钮的 onClick 事件不会起作用。需要插入浏览器端 React 来接管服务器静态标记渲染的位置。

创建 app.jsx 作为仅供浏览器使用的文件。它不会在服务器上执行(不像组件)。该文件是放置 ReactDOM.render() 调用以启用浏览器端 React 的地方：

```

ReactDOM.render(<MessageBoard messages={messages}/>,
  document.getElementById('message-board')
)

```

还需要使用全局 messages 作为 MessageBoard 的属性。messages 属性值由服务器端模板和 {{props}} 数据填充(参见 17.4.3 节)。换句话说，当模板从 Express 路由/中的 props 变量获取数据(也称为 locals)时，从 index.hbs 中填充消息的 messages 数组。

如果不能在服务器和浏览器上向 MessageBoard 提供相同的 messages 属性，将导致浏览器端 React 重新绘制整个组件，因为浏览器端 React 会考虑视图的不同。在底层，React 使用 checksum 属性将已经在 DOM(从服务器端渲染)中的数据与浏览器端 React 渲染的 checksum 进行比较。React 之所以使用 checksum，是因为它比做真实的 DOM 树比较要快(DOM 树比较会需要些时间)。

在 app.jsx 文件中，需要一些前端库，然后渲染 DOM 中的组件，参见代码清单



17.11(ch17/message-board/client/app.jsx)。

代码清单17.11 在浏览器上渲染React组件

```
const React = require('react')
const ReactDOM = require('react-dom')

const Header = require('../components/header.jsx')
const Footer = require('../components/footer.jsx')
const MessageBoard = require('../components/board.jsx')

ReactDOM.render(<Header />, document.getElementById('header'))
ReactDOM.render(<Footer />, document.getElementById('footer'))
ReactDOM.render(<MessageBoard messages={messages}/>,
  document.getElementById('message-board'))
```

浏览器端代码很少。

## 17.4.6 配置 Webpack

最后一步是设置 Webpack 以将浏览器代码打包到一个文件中、管理依赖关系和转换 JSX 代码。首先，需要做如下 Webpack 配置，入口点为 client/app.jsx，将输出设置为项目文件夹中的 public/js，并使用 Babel 加载器。将 devtool 设置为可以在 Chrome DevTools 中获得源代码行(不是编译后的代码行)：

```
module.exports = {
  entry: './client/app.jsx',
  output: {
    path: __dirname + '/public/js/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',
  stats: {
    colors: true,
    reasons: true
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
}
```

要将 JSX 转换为 JS，可以使用 babel-preset-react，并在 package.json 中指定 Babel 配置：

```
...
"babel": {
  "presets": [
```

```
    "react"
  ]
},
...
```

`package.json` 中的 Babel 和 Webpack 这样的客户端依赖(对于浏览器端 React)是开发环境的依赖包, 因为 Webpack 会把所有需要的东西打包到 `bundle.js` 中。因此, 在生产环境中运行时并不需要它们:

```
{
  ...
  "devDependencies": {
    "axios": "0.13.1",
    "babel-core": "6.10.4",
    "babel-jest": "13.2.2",
    "babel-loader": "6.2.4",
    "babel-preset-react": "6.5.0",
    "node-dev": "3.1.3",
    "webpack": "1.13.1"
  }
}
```

提示: 一定要使用这里提供的确切版本。否则, 当我完成这个段落的时候, 所有的东西都可能会破坏这个项目。

另外, 在 `package.json` 中添加如下 `npm` 构建脚本(这是可选的, 但使用更方便):

```
...
"scripts": {
  ...
  "build": "./node_modules/.bin/webpack"
},
...
```

我个人喜欢使用监听(`watch`)模式的 Webpack(`-w`)。在 `package.json` 中, 可以将选项 `-w` 添加到 `npm` 构建脚本中:

```
...
"scripts": {
  "build": "./node_modules/.bin/webpack -w",
  ...
},
...
```

因此, 每次运行 `npm run build` 时, Webpack 都会使用 Babel 将 JSX 转换成 JS, 并将所有依赖的文件拼接成一个文件。在本例中, 构建后的文件是 `public/js/bundle.js`。

由于构建后的文件包含在 `views/index.hbs` 模板中, 并在结束标签 `</body>` 之前, 因此浏览器端代码处于运行状态(以下代码行是模板中的内容):

```
<script type="text/javascript" src="/js/bundle.js"></script>
```

当用 `npm run build` 运行这个默认任务时，可以看到如下日志：

```
Hash: 1d4cfcb6db55f1438550
```

```
Version: webpack 1.13.1
```

```
Time: 733ms
```

Asset	Size	Chunks	Chunk Names
bundle.js	782 kB	0 [emitted]	main
bundle.js.map	918 kB	0 [emitted]	main
+ 200 hidden modules			

这是一个好的迹象。如果看到其他消息或错误，请将项目与 [www.manning.com/books/react-quickly](http://www.manning.com/books/react-quickly) 或 <https://github.com/azat-co/response-surgery/stud/master/ch17> 上的代码进行比较。

### 17.4.7 运行应用

运行应用就像在 Express 应用中渲染 React 组件。通常情况下，需要以下内容(前提是已经有一个构建过程和组件)：

- 输出未转义 locals/数据的模板。
- 调用 `res.render()` 来填充数据到模板并渲染数据(组件、属性等)。
- 将用于交互的浏览器端 React 文件(该文件含有 `ReactDOM.Render`)包含在模板中。

你是否仍然对同构的 Express 和 React 感到困惑？如果是这样，请从 [www.manning.com/books/react-quickly](http://www.manning.com/books/react-quickly) 或 <https://github.com/azat-co/react-quickly/tree/master/ch17/message-board> 获取项目的测试工作代码并浏览。可以删除 `app.jsx` 中的代码以禁用浏览器端 React(此时没有交互，例如鼠标单击)，或者删除 `index.js` 中的代码以禁用服务器端 React(加载页面时会略有延迟)。

要运行该项目，请运行 `MongoDB($ mongod`；更多信息可参考附录 D)。在项目文件夹中，运行如下命令：

```
$ npm install
```

```
$ npm start
```

不要忘记让 Webpack 以监听(`watch`)模式(`npm run build`)运行构建，或者每次更改浏览器代码时重新启动应用。

在浏览器中打开 `http://localhost:3000`，你会看到留言板(见图 17.9)。如果仔细观察页面加载的方式(Chrome DevTools)，你会看到第一个加载速度很快，因为 HTML 在服务器上渲染。

当在 `ch17/message-board/index.js` 中注释负责服务器端渲染的代码时，可以通过查看 Network 选项卡来比较时间。在该选项卡中，请注意 `localhost` 资源(第一次加载和服务器端渲染)和 `GET /messages` 的 XHR 调用。在我的结果中，`localhost` 要比 `GET /messages` 快得多，如图 17.10 所示。





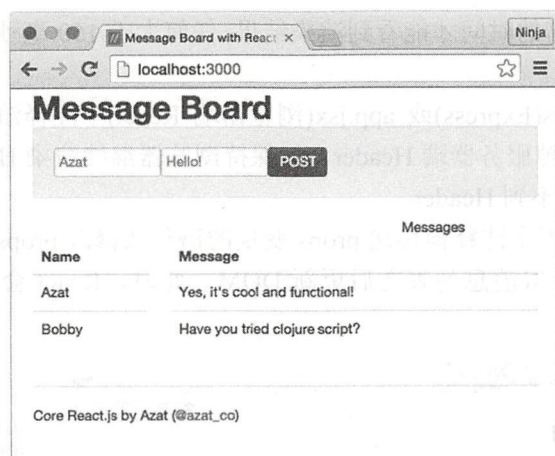
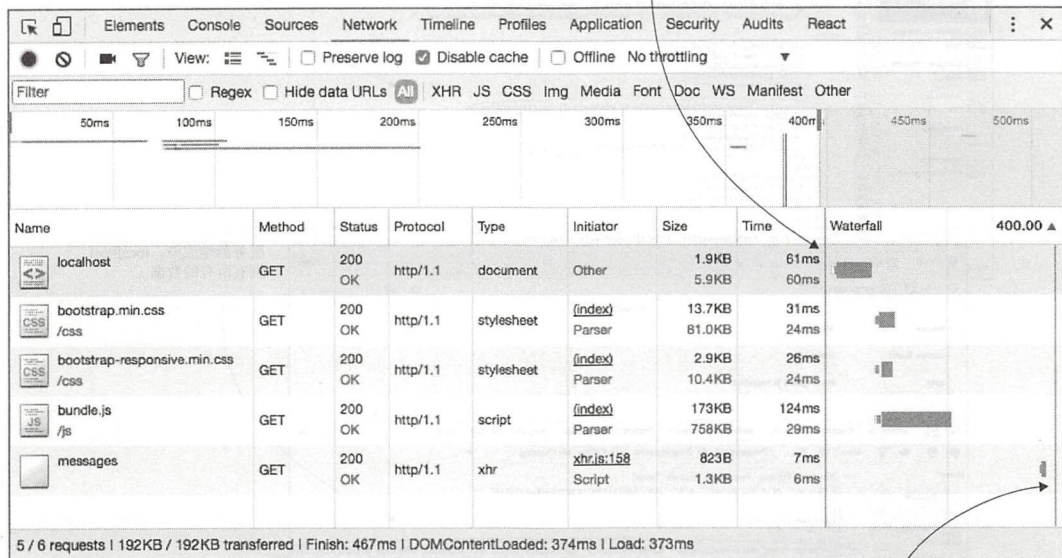


图 17.9 使用服务器和浏览器渲染的运行中的同构应用

完整的服务器端HTML(第一次加载)



如果没有完整的服务器端HTML, 那么用户必须等待调用这个XHR获取数据之后才能开始浏览器渲染

图 17.10 加载服务器端的 HTML 比完全加载要快 10 倍, 其中加载 bundle.js 比较慢

当然, 总的加载时间的大部分被 bundle.js 占用, 毕竟它有 200 多个模块。GET /messages 不会花太长时间, 只需要几毫秒。但是, 当 localhost 调用发生时, 用户就可以看到页面上的所有内容。相反, 如果没有同构代码, 用户将只能在 GET /messages 和浏览器端 React 渲染客户端 HTML 之后才能看到完整的 HTML。

让我们通过比较同构和浏览器渲染方式来从不同的角度看待这个应用。图 17.11 显示了 localhost 的结果。使用同构方法, localhost 拥有所有的数据, 而且仅需要 20 到 30 毫秒的时间来加载。对于只在浏览器端使用 React 相比, localhost 只有简单的骨架 HTML, 用



户不得不等待大约 10 倍的时间才能看到渲染结果。任何大于 150 毫秒的事情都是能被人类感知到的。

可以通过在 `index.js`(Express)或 `app.jsx`(浏览器端 React)中注释渲染语句来进行不同的尝试。例如,如果注释掉服务器端 Header,但保持浏览器继续渲染 Header,那么在出现之前可能会有片刻时间看不到 Header。

另外,如果在服务器上注释掉传递 `props` 变量的语句或修改 `props` 变量的值,浏览器端 React 会在通过 `axios` 获取消息列表之后更新 DOM。此时,React 会给出校验和不匹配的警告。

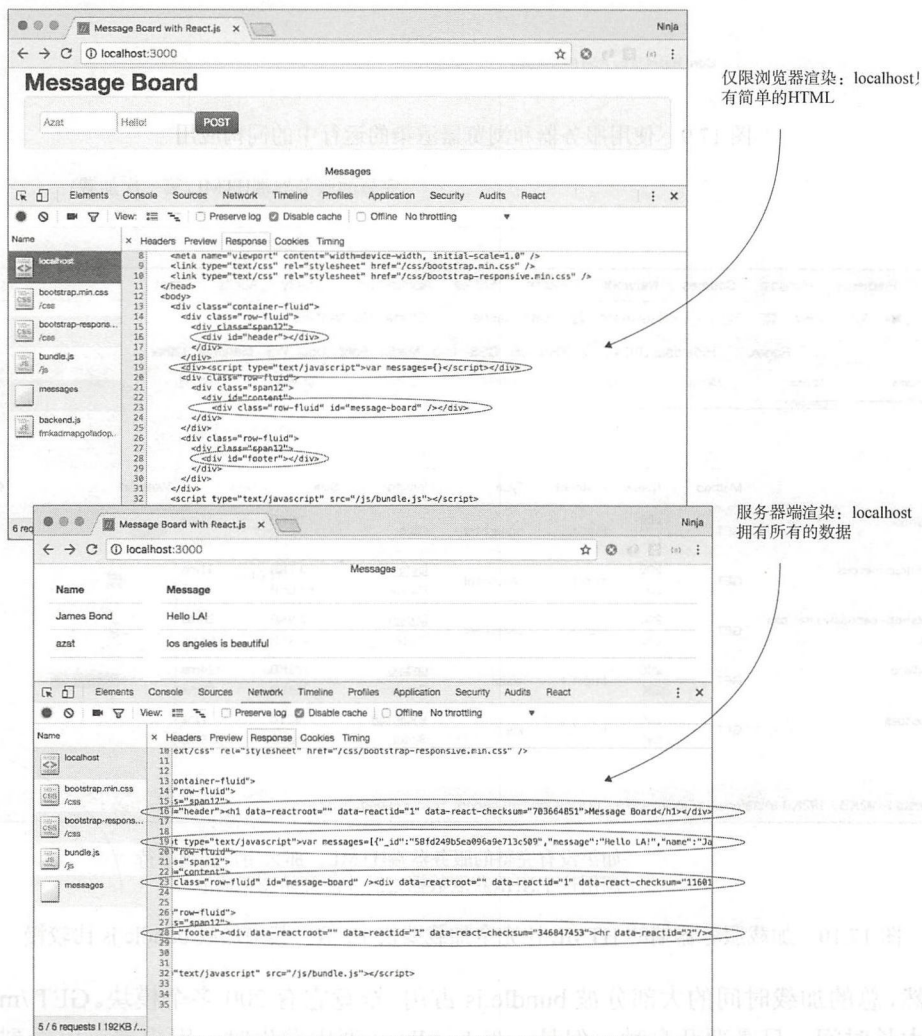


图 17.11 仅限浏览器渲染(上图)与服务器端渲染(下图)的 localhost(第一个响应)结果

### 同构路由和数据

应用迟早会增长,需要使用 React Router 和 Redux 这样的库来路由数据(在第 13 和第 14 章中有介绍)。有趣的是,这些库已经支持 Node,而且 React Router 甚至已经支持 Express。例如,可以将 React Router 路由传递到 Express,然后通过 `match` 和 `RouterContext` 来使服





务器端进行支持，从而在服务器端渲染组件：

```
const { renderToString } = require('react-dom/server')
const { match, RouterContext } = require('react-router')
const routes = require('./routes')

// ...
app.get('/', (req, res) => {
  match({ routes,
    location: req.url
  },
    (error,
    redirectLocation,
    renderProps) => {
    // ...
    res
      .status(200)
      .send(renderToString(
        <RouterContext {...renderProps} />
      ))
    })
})
```

使用 React Router 的一个特殊方法

将 location/URL 传递给 React Router 方法

使用特殊的 React Router 组件和属性渲染 HTML 字符串

Redux 有 `createStore()` 方法(第 14 章), 可以在 Express 中间件中使用服务器端提供数据存储。例如, 对于 App 组件, 使用 Redux 的服务器端代码如下所示:

```
const { createStore } = require('redux')
const { Provider } = require('react-redux')
const reducers = require('./modules')
const routes = require('./routes')

// ...
app.use((req, res) => {
  const store = createStore(reducers)
  const html = renderToString(
    <Provider store={store}>
      <App/>
    </Provider>
  )
  const preloadedState = store.getState()

  res.render('index', {html, preloadedState})
})
```

创建一个新的 Redux store 实例

将组件渲染为一个字符串

启用 store

从 Redux store 获取初始状态

使用 HTML 和数据渲染页面并返回给客户端

index 模板如下所示:

```
<div id="root">${html}</div>
<script>
  window.__PRELOADED_STATE__ = ${JSON.stringify(preloadedState)}
</script>
<script src="/static/bundle.js"></script>
```

Redux 使用与留言板相同的方法: 在 `<script>` 标记中渲染 HTML 和数据。

可以在 <http://mng.bz/F5pb> 和 <http://mng.bz/Edyx> 上查看带有说明的完整示例。





到这就结束了对同构 JavaScript 的讨论。同构 JavaScript 提供的一致性和代码重用有巨大的好处，可以帮助开发者更高效地工作，更快乐地生活！

## 17.5 测验

1. 用于在服务器上渲染 React 组件的方法是什么？
2. 在服务器上渲染首页可以提高性能。这么理解正确还是错误？
3. 对于 CommonJS 和 Node.js 模块语法，使用 `require()` (和 Webpack 一起)，可以在浏览器端代码中使用“require”来导入 npm 模块。这么理解正确还是错误？
4. 以下哪一项在 Handlebars 中用于输出非转义字符串？`<%...%>`、`{{...}}`、`{{{...}}}` 还是 `dangerouslySetInnerHTML = ...`？
5. 在浏览器端 React 中发起 AJAX/XHR 调用的最佳位置(而不会在服务器上触发)在哪里？

## 17.6 小结

- 要在服务器上使用和渲染 React，需要 `react-dom/server` 和 `renderToString()`。
- 数据必须相同才能将服务器端 React 渲染的 HTML 与浏览器端 React 渲染的 HTML 同步。React 使用校验和进行比较。
- `renderToString()` 和 `renderToStaticMarkup()` 之间的区别在于前者拥有校验和，允许浏览器端 React 重用 HTML，而后者不允许。
- 对于通用 JS 来说，在服务器上渲染 React，提供带有相同数据的浏览器端 React，并渲染浏览器端 React 组件。
- 使用三重花括号 `{{{html}}}` 在 Handlebars 模板中输出未转义的 HTML 内容。

## 17.7 测验答案

1. `ReactDOMServer.renderToString()`。`renderToStaticMarkup()` 不会渲染校验和。
2. 正确。可以在第一次加载所有数据，而无须等待 `bundle.js` 和 AJAX 请求。
3. 正确。Webpack 可以直接使用 `require()` 和 `module.exports` 语法。只需要在 `webpack.config.js` 中设置一个入口点，就可以让 Webpack 遍历所有依赖关系，而且只包含需要的依赖关系。
4. `{{{...}}}` 是正确的语法。对于转义变量，可以使用 `{{data}}` 以确保使用更安全。
5. `componentDidMount()`，因为它永远不会在服务器渲染上被调用。



# 第 18 章

## 使用 React Router 创建一个网上书店

### 本章内容：

- 项目结构和 Webpack 配置
- HTML 主页
- 创建组件
- 启动程序

本章中的项目主要侧重于演示如何使用 React Router 和一些 ES6 功能以及 Webpack。在这个项目中，你将创建一个简单的网上书店(见图 18.1)。

你将学习如何创建浏览器路由，以及使用 React Router 的以下技术：

- 如何传递数据到路由并访问它
- 如何访问 URL 参数
- 如何通过更改 URL 创建模态窗口
- 如何通过嵌套路由进行布局

为了说明这些技术，该项目包括如下不同的路由视图：

- 首页(/): 首页图书列表
- 产品页(/product/:id): 产品详情页
- 购物车(/cart): 显示用户选择的图书
- 结算(/checkout): 带有图书清单的收据



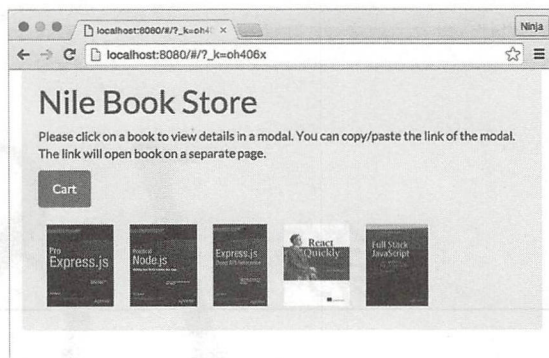


图 18.1 显示图书列表的 Nile 网上书店的首页

产品信息来自一个文件(ch18/nile/jsx/app.js, 参见下一节中的项目结构)中的一组数据。产品页可以作为模态对话框或单独的页面。当单击首页上的产品图片时, 将打开一个模态对话框; 例如, 图 18.2 显示了一个具有图书 *React Quickly* 详细视图的模态对话框。

URL `/products/3` 的后面是哈希值, 哈希值用来跟踪状态。这个链接是可以共享的: 如果新的窗口/选项卡中打开它, 它会是一个普通页面, 而不是一个模态对话框(见图 18.3)。当浏览列表, 并且不想跳转到丢失了上下文的新页面时, 模态是非常有用的。但是, 当打开共享的产品链接时, 并不需要上下文或列表, 而是需要将注意力集中在产品上。

创建网上书店的前端页面包括以下步骤:

- ① 使用 npm、Babel 和 Webpack 配置项目
- ② 创建 HTML 文件
- ③ 创建组件
- ④ 启动项目

我鼓励你去实现本章末尾“测验”部分列出的操作, 并将代码提交到本书的 GitHub 仓库: <https://github.com/azat-co/react-quickly>。

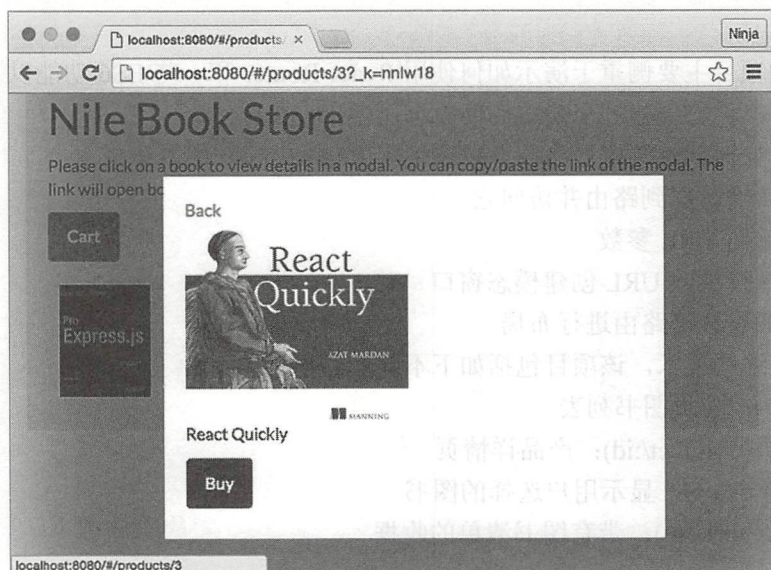


图 18.2 在模态窗口中展示产品视图







图 18.3 在新窗口中直接通过链接打开产品视图

注意：要跟进项目，需要下载未压缩版本的 React 并安装 Node.js 和 npm，用来编译 JSX。我也使用 Webpack 作为构建工具。附录 A 涵盖了安装这些工具的方法。

让我们从设置项目开始。

## 18.1 项目结构和 Webpack 配置

你应该对该项目的最终结果有一个基本的了解：一个带有 URL 路由的前端 Web 应用。直接跳入项目结构部分，文件目录结构如下：



为简洁起见，我省略了图片和 `node_modules` 文件夹的内容。这是一个前端应用，但是需要 `package.json` 来安装依赖，并告诉 Babel 需要做什么。代码清单 18.1 完整列出了 `package.json` 中的依赖：

代码清单18.1 Nile网上书店项目的依赖和设置

```
{
  "name": "nile",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "author": "Azat Mardan",
  "license": "MIT",
  "scripts": {
    "build": "node ./node_modules/webpack/bin/webpack.js -w"
  },
  "babel": {
    "plugins": [
      "transform-react-jsx"
    ],
    "presets": [
      "es2015"
    ],
    "ignore": [
      "js/bundle.js",
      "node_modules/**/*.js"
    ]
  },
  "devDependencies": {
    "babel-core": "6.3.21",
    "babel-loader": "6.4.1",
    "babel-plugin-transform-react-jsx": "6.3.13",
    "babel-preset-es2015": "6.3.13",
    "history": "4.0.0",
    "react": "15.5.4",
    "react-addons-test-utils": "15.2.1",
    "react-dom": "15.5.4",
    "react-router": "2.8.0",
    "webpack": "2.4.1",
    "webpack-dev-server": "1.14.0"
  }
}
```

创建使用watch模式构建代码的npm脚本

为Babel添加JSX插件

添加ES6/ES2015到ES5的转换(用来支持旧版浏览器)

不需要Babel编译的依赖

安装与React Router一起使用的history库

在标准项目属性之后是 `scripts` 命令，这个命令指向本地安装的 Webpack。这样可以保证正在使用的 Webpack 与 `devDependencies` 属性中的版本是一致的。构建 `bundle.js` 文件，并在端口 8080 上启动开发服务器。可以不使用它，而是在每次更改代码后手动构建，并使用 `node-static`(<https://github.com/cloudhead/node-static>)或类似的本地 Web 服务器：

```
"scripts": {
  "build": "node ./node_modules/webpack/bin/webpack.js -w"
},
```

下一行是 Babel v6.x 需要的配置，因为没有的话，Babel 就不会做太多的事情。如下配置告诉 Babel 使用 JSX 转换器和 ES2015 预设：

```
"babel": {
```

```

    "plugins": [
      "transform-react-jsx"
    ],
    "presets": [
      "es2015"
    ]
  },

```

如下 Babel 配置是可选的。它从 Babel 加载器中排除一些文件，如某些 `node_modules` 文件夹和文件：

```

    "ignore": [
      "js/bundle.js",
      "node_modules/**/*.js"
    ]
  },

```

注意：接下来定义依赖。需要使用确切的版本号，因为不能保证将来的版本可以正常工作。鉴于 React 与 Babel 的发展速度较快，它们很可能发生变化。但是使用稍微较旧的版本来学习概念没有任何问题，就像你在本书中所做的那样。

顾名思义，`devDependencies` 用于开发，并不是生产部署的一部分。这里是放置 Webpack、Webpack 开发服务器、Babel 和其他软件包的地方。请仔细检查是否使用了此处列出的确切版本：

```

...
"devDependencies": {
  "babel-core": "6.3.21",
  "babel-loader": "6.4.1",
  "babel-plugin-transform-react-jsx": "6.3.13",
  "babel-preset-es2015": "6.3.13",
  "history": "4.0.0",
  "react": "15.5.4",
  "react-addons-test-utils": "15.2.1",
  "react-dom": "15.5.4",
  "react-router": "2.8.0",
  "webpack": "2.4.1",
  "webpack-dev-server": "1.14.0"
}

```

现在已经定义了项目依赖，接下来需要设置 Webpack 的构建过程，以便可以使用 ES6 和转换 JSX。为此，请使用代码清单 18.2(ch18/nile/webpack.config.js)在根目录中创建文件 `webpack.config.js`。

#### 代码清单 18.2 Nile 网上书店的 Webpack 配置

```

module.exports = {
  entry: './jsx/app.jsx',
  output: {
    path: __dirname + '/js',

```



```
    filename: "bundle.js"
  },
  devtool: '#sourcemap',
  stats: {
    colors: true,
    reasons: true
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
}
```

使用 `npm i`(`npm install` 的缩写)安装依赖,至此就完成了项目的设置。接下来创建 HTML 文件,该文件包含一个用来挂载 React 组件的 `<div>` 元素。

## 18.2 HTML 主页

这个项目的 HTML 非常基础,它有一个包含 id 为 `content` 的容器,并且引入 `js/bundle.js` 文件,如代码清单 18.3 所示(ch18/nile/index.html)。

代码清单18.3 主机HTML文件

```
<!DOCTYPE html>
<html>
  <head>
    <link href="css/bootstrap.css" type="text/css" rel="stylesheet"/>
  </head>
  <body>
    <div class="container-fluid">
      <div id="content" class=""></div>
    </div>
    <script src="js/bundle.js"></script>
  </body>
</html>
```

现在可以做快速测试,看看构建和开发过程是否正常工作:

- ① 使用 `$ npm install` 安装所有依赖。只需要执行一次。
- ② 将 `console.log('Hey Nile!')` 写入 `jsx/app.jsx`。
- ③ 使用 `npm run build` 运行应用。之后应用会保持运行, `-w` 会在代码更改时自动重新构建文件。
- ④ 从项目根目录启动本地 Web 服务器。可以使用 `node-static` 或 `webpack-dev-server`

(package.json 已经包含)。

⑤ 在浏览器中打开 <http://localhost:8080>。

⑥ 打开浏览器控制台(如 Chrome DevTools)。应该能看到“Hey Nile!”这条消息。

## 18.3 创建组件

继续构建应用程序(假设能看到“Hey Nile!”消息)。你将开始使用 ES6 模块化和解构导入模块。简单地说,解构是通过使用与对象的属性之一相同的名称从对象定义变量的一种方法。例如,如果要从 `user.accounts` 导入 `accounts` 并声明 `accounts`, 可以使用 `{accounts} = user`。如果不了解解构, 请参阅附录 E 中的 ES6 速查表。

### 18.3.1 主文件: app.jsx

编写第一个文件 `app.jsx`, 可以在其中设置导入依赖、图书信息和路由。

先去除组件代码, 稍后会单独讲解组件代码, `app.jsx`(`ch18/nile/jsx/app.jsx`)中的代码如代码清单 18.4 所示。

代码清单 18.4 主应用文件

```
const React = require('react')
const ReactDOM = require('react-dom')
const { hashHistory, Router, Route, IndexRoute, Link, IndexLink } = require('react-router')
const Modal = require('./modal.jsx')
const Cart = require('./cart.jsx')
const Checkout = require('./checkout.jsx')
const Product = require('./product.jsx')

const PRODUCTS = [
  { id: 0, src: 'images/proexpress-cover.jpg', title: 'Pro Express.js', url: 'http://amzn.to/1D6qiqk' },
  { id: 1, src: 'images/practicalnode-cover.jpeg', title: 'Practical Node.js', url: 'http://amzn.to/NuQ0fM' },
  { id: 2, src: 'images/expressapiref-cover.jpg', title: 'Express API Reference', url: 'http://amzn.to/lxcHanf' },
  { id: 3, src: 'images/reactquickly-cover.jpg', title: 'React Quickly', url: 'https://www.manning.com/books/react-quickly' },
  { id: 4, src: 'images/fullstack-cover.png', title: 'Full Stack JavaScript', url: 'http://www.apress.com/9781484217504' }
]

const Heading = () => {
  return <h1>Nile Book Store</h1>
}

const Copy = () => {
```

导入 hashHistory

从 React Router 导入对象

导入组件

图书数据数组, 所以在这个例子中不需要使用数据库

这些组件是无状态组件

```

    return <p>Please click on a book to view details in a modal. You can
    ➡ copy/paste the link of the modal. The link will open the book on a
    ➡ separate page.</p>
  }

```

```

class App extends React.Component {
  ...
}

```

```

class Index extends React.Component {
  ...
}

```

```

let cartItems = {}
const addToCart = (id) => {
  if (cartItems[id])
    cartItems[id] += 1
  else
    cartItems[id] = 1
}

```

cartItems对象保存购物车中的商品。初始化为空

```

ReactDOM.render((
  <Router history={hashHistory}>
    <Route path="/" component={App}>
      <IndexRoute component={Index}/>
      <Route path="/products/:id" component={Product}
        addToCart={addToCart}
        products={PRODUCTS} />
    </Route>
    <Route path="/cart" component={Cart}
      cartItems={cartItems} products={PRODUCTS}/>
    </Route>
    <Route path="/checkout" component={Checkout}
      cartItems={cartItems} products={PRODUCTS}/>
    </Route>
  ), document.getElementById('content'))

```

在文件顶部导入所有内容后，硬编码产品数据为数组，数组中的每个对象都有 id、src、title 和 url 属性。很明显，在真实项目中需要从服务器获取这些数据，而不是从浏览器 JavaScript 文件中获取：

```

const PRODUCTS = [
  { id: 0, src: 'images/proexpress-cover.jpg',
    title: 'Pro Express.js', url: 'http://amzn.to/1D6qiqk' },
  { id: 1, src: 'images/practicalnode-cover.jpeg',
    title: 'Practical Node.js', url: 'http://amzn.to/NuQ0fM' },
  { id: 2, src: 'images/expressapiref-cover.jpg',
    title: 'Express API Reference', url: 'http://amzn.to/1xcHanf' },
  { id: 3, src: 'images/reactquickly-cover.jpg',
    title: 'React Quickly',
    url: 'https://www.manning.com/books/react-quickly'},
  { id: 4, src: 'images/fullstack-cover.png',
    title: 'Full Stack JavaScript',
    url: 'http://www.apress.com/9781484217504' }
]

```

使用 ES6 箭头函数实现无状态组件 Heading。为什么不直接渲染<h1>？因为这样做，就可以在多个组件上使用它。使用相同的方式定义无状态组件 Copy。它只是静态的 HTML，所以不需要任何额外的东西，甚至不需要属性：

```

const Heading = () => {

```



```

return <h1> Nile Book Store</h1>
}

const Copy = () => {
  return <p>Please click on a book to view details in a modal. You can
    ➡ copy/paste the link of the modal. The link will open the book on a
    ➡ separate page.</p>
}

```

接下来是 App 和 Index 这两个主要组件，后面是 cartItems 对象，cartItems 对象保存了购物车中的当前项。cartItems 初始化为空，addToCart() 只实现了如下简单的功能：在服务器端版本中，可以使用 Redux 将数据保存到服务器和会话中，以便用户稍后可以回到购物车。

```

let cartItems = {}
const addToCart = (id) => {
  if (cartItems[id])
    cartItems[id] += 1
  else
    cartItems[id] = 1
}

```

最后，是用于挂载 Router 组件的 ReactDOM.render() 方法。需要将 history 库传递给 React Router。正如前面提到的，可以是 browserHistory 或 hashHistory(本项目使用后者)：

传递一个方法，用来  
添加图书到购物车中

```

ReactDOM.render((
  <Router history={hashHistory}>
    <Route path="/" component={App}>
      <IndexRoute component={Index}/>
      <Route path="/products/:id" component={Product}
        addToCart={addToCart}
        products={PRODUCTS} />
      <Route path="/cart" component={Cart}
        cartItems={cartItems} products={PRODUCTS} />
    </Route>
    <Route path="/checkout" component={Checkout}
      cartItems={cartItems} products={PRODUCTS} />
  </Router>
), document.getElementById('content'))

```

在 IndexRoute 中  
使用 Index 组件

将购物车中的产品列表和所有产品列表分  
别作为 cartItems 和 products 属性

在 App 组件外定义结算组  
件，避免渲染标题

对于 /products/:id 路由，Product 组件路由获取 addToCart() 函数，用来将书目加入购物车。该函数将在 this.props.route.addToCart 中可用，因为传递给 Route 的任何属性需要在组件的 this.props.route.NAME 中获取。例如，在组件 Product 中，products 通过 this.props.route.products 获取：

```

<Route path="/products/:id" component={Product} addToCart={addToCart}
  products={PRODUCTS} />

```

/checkout 路由在 App 组件之外，所以它没有标题(见图 18.4)。是否记得，path 和路由结构是可以独立的：

```
<Route path="/checkout" component={Checkout}
  cartItems={cartItems} products={PRODUCTS}/>
```



图 18.4 另一个不显示标题的结算单视图

在本例中，通过将 Checkout(结算)组件放在 App 组件之外，Checkout 就不会是 App 的子组件。可以单击 Back 按钮，从结算页面(/checkout)返回上一页面。

### App 组件

现在可以实现 App 组件了！它是主组件，因为它是 Webpack 的入口点，并且为大多数视图提供了布局：渲染子组件(如 Product、产品列表和 Cart)，显示模态对话框。记住 ReactDOM.render()，它非常重要，它表明 App 是应用程序的根组件：

```
ReactDOM.render((
  <Router history={hashHistory}>
    <Route path="/" component={App}>           ← App是Product、Cart和
      <IndexRoute component={Index}/>             Index的祖先节点
      <Route path="/products/:id" component={Product} .../>
    </Route>
    // ...
  </Router>
), document.getElementById('content'))
```

与无状态组件不同，无状态组件仅仅是函数，App 组件才是真正的“组件”，如代码清单 18.5 所示(ch18/nile/jsx/app.jsx)。

### 代码清单18.5 App组件

```
class App extends React.Component {
  componentWillReceiveProps(nextProps) {
    this.isModal = (nextProps.location.state &&
      nextProps.location.state.modal)
    if (this.isModal &&
      nextProps.location.key !== this.props.location.key) {
```

使用Link中传递的状态(在Route中实现)

```

    this.previousChildren = this.props.children
  }
}
render() {
  console.log('Modal: ', this.isModal)
  return (
    <div className="well">
      <Heading/>
      <div>
        {(this.isModal) ? this.previousChildren :
          this.props.children}

        {(this.isModal)?
          <Modal isOpen={true} returnTo=
            {this.props.location.state.returnTo}>
            {this.props.children}
          </Modal> : ''
        }
      </div>
    </div>
  )
}
}

```

保存以前的子节点

如果处在模态状态，展示旧的子节点，否则展示路由中定义的子节点

使用模态窗口展示产品详情

`componentWillReceiveProps()` 接收最新属性作为参数，在这个方法中确定此视图是否处在模态状态：

```

class App extends React.Component {
  componentWillReceiveProps(nextProps) {
    this.isModal = (nextProps.location.state &&
      nextProps.location.state.modal)
  }
}

```

以下条件用于检查是否是模态页面。如果是模态页面，将保存子节点。`isModal` 布尔值根据在 `Link` 元素(在 `Index` 组件中有示例)中设置的 `location.state` 来确定页面是否处在模态：

```

if (this.isModal &&
  nextProps.location.key !== this.props.location.key) {
  this.previousChildren = this.props.children
}
}

```

在 `render()` 函数中，`Heading` 是否只是一个函数(无状态组件)并不重要。可以像任何其他 `React` 组件一样渲染它：

```

render() {
  console.log('Modal: ', this.isModal)
  return (
    <div className="well">
      <Heading/>
    </div>
  )
}

```

使用三元运算符判断渲染的是 `this.previousChildren` 还是 `this.props.children`。`React Router` 会从其他嵌套路由/组件(如 `Index` 和 `Product`)填充 `this.props.children`。请记住，几乎所有的页面都使用了 `App` 组件。默认情况下，在 `React Router` 中需要渲染 `this.props.children`：

```

<div>

```



```
{(this.isModal) ? this.previousChildren: this.props.children}
```

如果没有 `isModal` 条件, 并且每次输出 `this.props.children`, 那么当单击图像来打开模态窗口时, 将始终看到相同的内容, 如图 18.5 所示。显然, 这个行为并不是你想要的。出于这个原因, 需要渲染以前的子节点(`previousChildren`), 在模态窗口的情况下这个 `previousChildren` 就是首页。可以重复使用 `state.modal` 为 `true` 的模态链接(稍后在 `Index` 组件中演示)。

最后, 可以在另一个三元运算符中渲染模态页面。在 `Modal` 组件中传递 `isOpen` 和 `returnTo` 属性:

```
{(isModal)?
  <Modal isOpen={true} returnTo={this.props.location.state.returnTo}>
    {this.props.children}
  </Modal> : ''
}
</div>
</div>
)
}
}
```

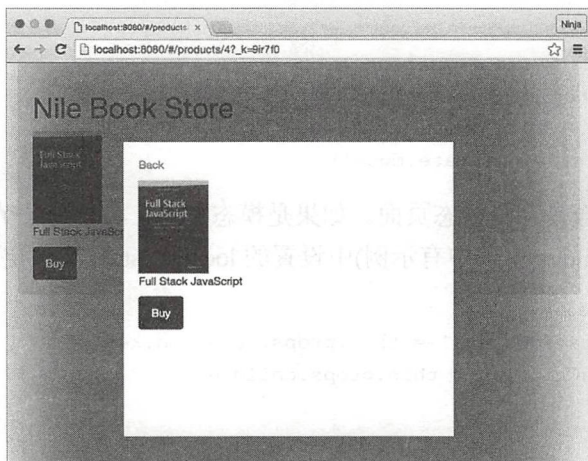


图 18.5 如果没有 `isModal` 和 `previousChildren`, 产品列表不会显示

## Index 组件

继续使用 `nile/jsx/app.jsx` 文件, 下一个组件是首页组件。是否记得, 首页组件显示完整的图书列表, 如代码清单 18.6 所示(`ch18/nile/jsx/app.jsx`):

代码清单18.6 首页的Index组件

```
class Index extends React.Component {
  render() {
    return (
      <div>
        <Copy/>
        <p><Link to="/cart" className="btn btn-danger">Cart</Link></p>
      </div>
    )
  }
}
```

添加Link链接  
接到购物车

```

    {PRODUCTS.map(picture => (
      <Link key={picture.id}
        to={{pathname: `/products/${picture.id}`,
          state: { modal: true,
            returnTo: this.props.location.pathname }
        }}
      >
        <img style={{ margin: 10 }} src={picture.src} height="100" />
      </Link>
    ))}
  </div>
</div>
)
}
}

```

显示模态窗口

使用ES6字符串模板创建产品URL

在 `map()` 迭代器中，渲染打开图书模态窗口的链接。当单独打开这些链接时，不会在模态窗口中展示：

```

{PRODUCTS.map(picture => (
  <Link key={picture.id}
    to={{pathname: `/products/${picture.id}`,
      state: { modal: true,
        returnTo: this.props.location.pathname }
    }}
  >
  </Link>
)}
)
}
}

```

可以将任何属性传递到与 `/products/:id` 路由关联的组件(即 `Product` 及其父组件 `App`)。可以在 `this.props.location.NAME` 中访问属性，其中 `NAME` 是属性的名称。你已经在 `Modal` 组件中使用过 `state.modal`。

`<img>` 标签使用 `src` 属性来加载图书图片：

```

    <img style={{ margin: 10 }} src={picture.src} height="100" />
  </Link>
)}
</div>
</div>
)
}
}

```

以上就是 `app.jsx` 文件。下一个要实现的组件是 `Cart` 组件，它处在自己的文件中，因为它与用于应用布局的 `App` 组件没有密切关系。

### 18.3.2 Cart 组件

`Cart` 组件渲染 `/cart` 路由，该组件用于显示购物车中的图书列表及图书数量，如图 18.6 所示。`Cart` 组件使用 `cartItems` 来获取图书列表及图书数量。注意 `render()` 函数使用 ES6 的风格，如代码清单 18.7 所示(`nile/jsx/cart.jsx`)。

## 代码清单18.7 Cart组件

```
const React = require('react')
const {
  Link
} = require('react-router')

class Cart extends React.Component {
  render() {
    return <div>
      {(Object.keys(this.props.route.cartItems).length == 0) ?
        <p>Your cart is empty</p> : ''
      }
      <ul>
        {(Object.keys(this.props.route.cartItems).map((item,
          index,
          list)=>{ ← 迭代并渲染购物车中的每一项
            return <li key={item}>
              {this.props.route.products[item].title}
              - {this.props.route.cartItems[item]}
            </li>
          })}
      </ul>
      <Link to="/checkout"
        className="btn btn-primary"> ← 添加导航到结账页面的
        Checkout                               按钮，结账页面显
                                              示结算清单
      </Link>
      <Link to="/" className="btn btn-info"> ← 添加导航到首页的按钮，让用
        Home                                     户购买更多产品
      </Link>
    </div>
  }
}
```

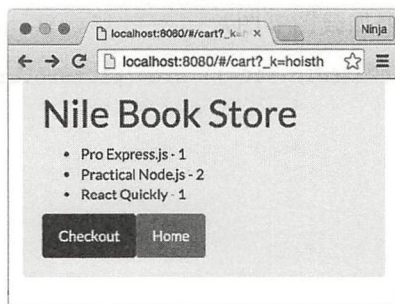


图 18.6 购物车

因为在 `app.jsx` 中定义路由时传递了 `products` 属性，所以 `Cart` 组件能从 `this.props.route.products` 获取产品(图书)列表：

```
<Route path="/cart" component={Cart}
  cartItems={cartItems} products={PRODUCTS}/>
```

如果使用 `Redux`(参见第 14 章)，则不需要手动传递属性(如 `products`)，因为 `Provider` 组件在子组件中自动填充在数据存储中存储的数据。



### 18.3.3 Checkout 组件

下一个组件是 Checkout 组件，如图 18.7 所示。这是唯一一个在 App 路由外的组件。回顾一下，如下是来自 app.jsx 的路由：

```
ReactDOM.render(() => {
  <Router history={hashHistory}>
    <Route path="/" component={App}>
      <IndexRoute component={Index}/>
      <Route path="/products/:id" component={Product}
        addToCart={addToCart}
        products={PRODUCTS} />
      <Route path="/cart" component={Cart}
        cartItems={cartItems} products={PRODUCTS} />
    </Route>
    <Route path="/checkout" component={Checkout}
      cartItems={cartItems} products={PRODUCTS} />
  </Router>
), document.getElementById('content'))
```

App路由：主布局

App路由之外的 Checkout路由

可以看到，App 和 Checkout 组件位于层次结构相同的级别。因此，当导航到/checkout 时，App 路由不会触发。有趣的是，可以通过嵌套 URL 将组件保留在嵌套结构之外：例如设置路由/cart/checkout。不过，你不会在这里这样做。

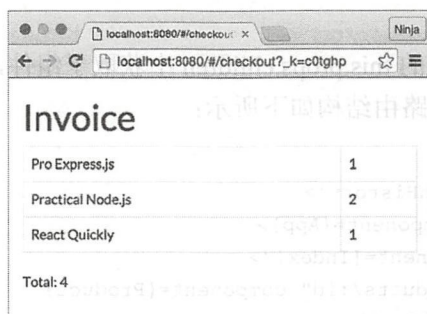


图 18.7 Checkout 组件不需要书名

使用 Twitter Bootstrap 表格和 table-bordered 展示结算清单。再次使用 ES6 的 const(记住，对象属性可以更改)和函数语法，参见代码清单 18.8(nile/jsx/checkout.jsx)。

#### 代码清单18.8 Checkout组件

```
const React = require('react')
const {
  Link
} = require('react-router')

class Checkout extends React.Component {
  render() {
    let count = 0
    return <div><h1>Invoice</h1><table className="table table-bordered">
      <tbody>
        {Object.keys(this.props.route.cartItems).map((item, index,
```

```

    list)=>{
      count += this.props.route.cartItems[item]
      return <tr key={item}>
        <td>{this.props.route.products[item].title}</td>
        <td>{this.props.route.cartItems[item]}</td>
      </tr>
    }
  </tbody></table><p>Total: {count}</p></div>
}
}

module.exports = Checkout

```

← 迭代并渲染购物车中的一个项

← 使用在路由中传递的图书列表来获取书名

← 导出类

接下来需要实现 Modal 组件。

### 18.3.4 Modal 组件

这个组件将自身的子组件渲染在模态对话框中。回想一下，在 App 组件中，以如下方式使用 Modal 组件：

```

({this.isModal) ?
  <Modal isOpen={true} returnTo={this.props.location.state.returnTo}>
    {this.props.children}
  </Modal> : ''
}

```

Modal 组件从 App 组件的 `this.props.children` 中获取子组件，该子组件在 `app.jsx` 的路由中定义并传递。再次提醒，路由结构如下所示：

```

ReactDOM.render((
  <Router history={hashHistory}>
    <Route path="/" component={App}>
      <IndexRoute component={Index}/>
      <Route path="/products/:id" component={Product}
        addToCart={addToCart}
        products={PRODUCTS} />
      <Route path="/cart" component={Cart}
        cartItems={cartItems} products={PRODUCTS}/>
    </Route>
    <Route path="/checkout" component={Checkout}
      cartItems={cartItems} products={PRODUCTS}/>
  </Router>
), document.getElementById('content'))

```

这样就可以通过独立页面或模态页面浏览图书详情。通过 URL 可知，嵌套在 App 路由下的组件就是 Modal 的子组件，如代码清单 18.9 所示(`nile/jsx/modal.jsx`)。

#### 代码清单 18.9 Modal 组件

```

const React = require('react')
const {
  Link
} = require('react-router')

```

```

class Modal extends React.Component {
  constructor(props) {
    super(props)
    this.styles = {
      position: 'fixed',
      top: '20%',
      right: '20%',
      bottom: '20%',
      left: '20%',
      width: 450,
      height: 400,
      padding: 20,
      boxShadow: '0px 0px 150px 130px rgba(0, 0, 0, 0.5)',
      overflow: 'auto',
      background: '#fff'
    }
  }
  render() {
    return (
      <div style={this.styles}>
        <p>
          <Link to={this.props.returnTo}>
            Back
          </Link>
        </p>
        {this.props.children}
      </div>
    )
  }
}

module.exports = Modal

```

← 将样式定义为类的实例属性

使用fixed定位在分离模式中将模态窗口浮动在视图中间

注意采用驼峰命名规范的boxShadow是CSS中的box-shadow

应用样式来创建模态视图

因为 Product 组件嵌套在 App 组件下, 而且路由/product/:id 对应 Product 组件, 在 Index 组件(图书列表)中设置 state.modal 为 true, 所以模态窗口将显示单独的 Product 组件。

### 18.3.5 Product 组件

Product 组件使用路由中的属性来触发操作(this.props.route.addToCart)。app.js 中的 addToCart() 方法将特定的书放在购物车中(如果使用 Redux, 就分发这个添加操作)。使用浏览器 onClick 事件处理程序和 Product 组件中的 handleBuy() 方法触发 addToCart(), 然后 app.js 中的 addToCart() 方法被调用。整个事件的顺序为: onClick -> this.handleBuy -> this.props.route.addToCart -> addToCart()(app.js)。提醒一下, addToCart() 中的代码如下:

```

let cartItems = {}
const addToCart = (id) => {
  if (cartItems[id])
    cartItems[id] += 1
  else
    cartItems[id] = 1
}

```

当然, 如果使用 Redux 或 Relay, 那么可以使用它们提供的方法。该例使用简单的数组作为数据存储和单个方法来保持简单。

现在来看看 Product 组件本身。和以往一样, 首先导入 React 并定义类, 然后是事件和



render()方法；代码清单 18.10 中是产品(nile/jsx/product.jsx)的完整代码，并指出了里面最有趣的部分。

代码清单18.10 单个产品信息

```
const React = require('react')
const {
  Link
} = require('react-router')

class Product extends React.Component {
  constructor(props) {
    super(props)
    this.handleBuy = this.handleBuy.bind(this)
  }
  handleBuy (event) {
    this.props.route.addToCart(this.props.params.id)
  }
  render() {
    return (
      <div>
        <img src={this.props.route.products[
          this.props.params.id].src}
          style={{ height: '80%' }} />
        <p>{this.props.route.products[this.props.params.id].title}</p>
        <Link
          to={{
            pathname: `/cart`,
            state: { productId: this.props.params.id}
          }}
          onClick={this.handleBuy}
          className="btn btn-primary">
          Buy
        </Link>
      </div>
    )
  }
}

module.exports = Product
```

bind(this)确保  
this指向正确

将图书id传递给  
app.jsx中的函数

通过图书id获取产品  
图片路径和文件名

单击Buy按钮时触  
发handleBuy函数

也可以在 Link 组件中给 Cart 组件传递状态：

```
<Link
  to={{
    pathname: `/cart`,
    state: { productId: this.props.params.id}
  }}
  onClick={this.handleBuy}
  className="btn btn-primary">
  Buy
</Link>
```

回想一下，模态间接使用 Product 组件：Modal 不渲染 Product，而是使用具有 Product 的 this.props.children。因此，Modal 可以被认为是透传组件(有关 this.props.children 和透传组件的更多信息，参见第 8 章)。

## 18.4 启动项目

这就是创建网上书店的全部内容。你已经使用了一些 ES6 功能，并通过 React Router 传递状态。现在通过使用 `npm run build` 来构建项目，启动本地 Web 服务器(WDS 或 `node-static`)，并导航到 `http://localhost:8080/nile`，前提是静态 Web 服务器运行在 `nile` 文件夹的父文件夹中(URL 路径取决于启动静态 Web 服务器的位置)。

你看到的首页上应该显示了网格布局的图书封面列表。当单击图书封面时，会出现一个模态窗口，单击 Buy 按钮将图书添加到购物车中(可通过路由 `/cart` 和 `/checkout` 查看购物车中的图书)。

## 18.5 测验

为了能有更多收获，不妨尝试下面的操作：

- 抽象(复制/粘贴)Index 和 App 到单独的文件中，并将 App 重命名为 Layout。
- 对数据使用永久存储(如 MongoDB 或 PostgreSQL)。
- 通过定制 Express 服务器，使用无哈希路由替换哈希路由。无哈希路由可参考第 15 章的 Netflix 克隆版项目。
- 使用 Jest 为 Product 和 Checkout 添加单元测试。

将代码提交到本书 GitHub 仓库(<https://github.com/azat-co/react-quickly/>)中 ch18 目录下的新文件夹中。

## 18.6 小结

- Link 组件可从 `react-router` 中导出，可以用来传递状态，例如 `<Link to={{pathname: '/product', state: {modal: true}}} >`。
- React Router 状态可在 `this.props.location.state` 中获取。
- 可以通过 `<Route name={value}>` 方式传递属性，并通过 `this.props.route.name` 方式获取。

# 第 19 章

## 使用 Jest 测试密码

### 本章内容：

- 项目结构和 Webpack 配置
- HTML 主页
- 实现强密码模块
- 创建 Jest 测试
- 实现 Password 组件和 UI

这个项目的重点是构建 UI、处理模块化、使用 Jest 进行测试以及介绍一些其他 React 相关技术，如组件构成、ES6 语法、状态、属性等。回想一下在第 16 章中涉及的测试：使用密码小部件作为单元测试和 UI 测试的示例。在本章的项目中，将构建小部件自身来检查、验证和生成新的密码。在这个过程中，将以扩展的方式再次解释测试。

密码小部件有默认禁用的 Save 按钮，当密码足够强(符合预设规则)时，这个按钮将被启用，如图 19.1 所示。另外，Generate 按钮可以创建强(根据规则)密码。对于已满足的规则，将会用横线划掉。此外，还有一个用于决定是否显示密码的复选框，就像大多数 Mac OS 界面一样(见图 19.2)。

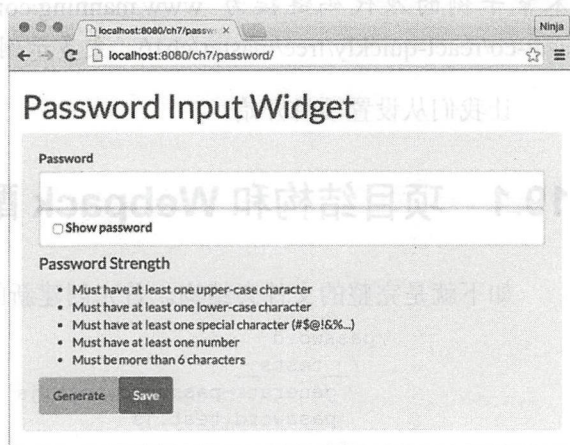


图 19.1 密码小部件：包含密码输入和自动生成符合规范的密码



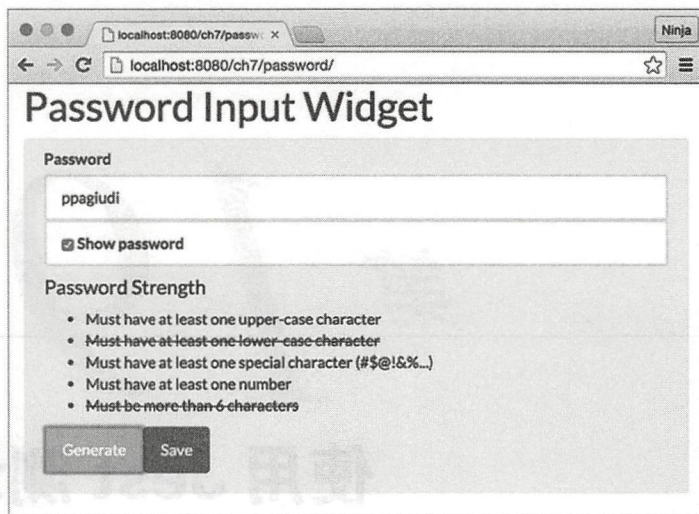


图 19.2 满足部分条件以及密码可见

父组件名为 Password，子组件如下：

- PasswordInput: 密码输入框
- PasswordVisibility: 决定密码显隐状态的复选框
- PasswordInfo: 保存密码前需要满足的规则列表
- PasswordGenerate: 生成满足所有规则的密码的按钮

密码小部件是使用单个父组件构建的。将密码强度规则提供给组件作为组件的属性，所以组件是高度可定制的。可以进行一些定制并在自己的应用中使用！

注意：要完成这个项目，需要安装 Node.js 和 npm 来编译 JSX。本例仍然使用 Webpack 作为构建工具，当然也使用 Jest 作为测试引擎。附录 A 介绍了如何安装这些工具。

注意：因为这个项目的一部分是在第 16 章中完成的，所以源代码在 ch16 文件夹中。本章示例的源代码链接为 [www.manning.com/books/react-quickly](http://www.manning.com/books/react-quickly) 和 <https://github.com/azat-co/react-quickly/tree/master/ch16>。更多示例请访问 <http://reactquickly.co/demos>。

让我们从设置项目开始。

## 19.1 项目结构和 Webpack 配置

如下就是完整的文件夹结构。首先创建新的名为 password 的项目文件夹：

```
/password
  /__tests__
    generate-password.test.js
    password.test.js
  /css
    bootstrap.css
  /dist
```

这个文件夹保存所有  
的Jest测试文件

保存Webpack构建  
后的文件的文件夹

```

    bundle.js
    bundle.js.map
  /js
    generate-password.js
    rules.js
  /jsx
    app.jsx
    password-generate.js
    password-info.jsx
    password-input.jsx
    password-visibility.jsx
    password.jsx
  /node_modules
    ...
  index.html
  package.json
  README.md
  webpack.config.js

```

项目入口文件 →

← 负责生成随机密码的库

← Webpack 配置文件

`__tests__` 文件夹用于 Jest 测试。css 文件夹包含名为 Flatly(<https://bootswatch.com/flatly>) 的 Twitter Bootstrap 主题。js 和 jsx 文件夹分别对应库和组件，其中 js/generate-password.js 是负责生成随机密码的库。

`dist` 文件夹包含带有源映射的已编译的 JSX 文件。这就是 Webpack 将会放置连接的文件及其 Source Map 文件的地方。在这里，`dist` 是 `distribution` 的缩写，是一个常用名称，除了 `dist` 之外，常用名称还有 `js` 或 `build` 等。这里使用 `dist` 来介绍一些种类，并介绍如何定制 Webpack 配置。

为了避免手动安装每个固定版本的依赖，可以将 `package.json` 从代码清单 19.1 复制到密码文件夹，然后在文件夹中运行 `npm install(ch16/password/package.json)`。

代码清单19.1 项目的依赖和设置

```

{
  "name": "password",
  "version": "2.0.0",
  "description": "",
  "main": "index.html",
  "scripts": {
    "test": "jest",
    "test-watch": "jest --watch",
    "build-watch": "./node_modules/.bin/webpack -w",
    "build": "./node_modules/.bin/webpack"
  },
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"
    ]
  },
  "devDependencies": {
    "babel-core": "6.10.4",
    "babel-loader": "6.4.1",
    "babel-preset-react": "6.5.0",
    "jest-cli": "19.0.2",
    "react": "15.5.4",

```

← 创建使用watch模式构建代码的npm脚本

← 在Jest中使用Babel来支持JSX

← 推荐本地安装Jest模块

```

    "react-test-renderer": "15.5.4",
    "react-dom": "15.5.4",
    "webpack": "2.4.1"
  }
}

```

← 使用react-test-renderer  
进行浅渲染

最有趣的是 `scripts` 部分，在这里我们进行测试、编译和打包：

```

"scripts": {
  "test": "jest",
  "test-watch": "jest --watch",
  "build-watch": "./node_modules/.bin/webpack -w",
  "build": "./node_modules/.bin/webpack"
},

```

回想一下，在第 18 章的 Nile 网上书店项目中使用了 `transform-react-jsx`：

```

"babel": {
  "plugins": [
    "transform-react-jsx"
  ],

```

但在这个项目中，我们使用 `React` 预设。这只是另一种完成相同事情的方法。可以使用预设或插件。不过预设是一种更现代化的方法并能用于更多的文档和项目。

测试脚本(`npm test`)用于手动运行 `Jest` 测试。相反，`test-watch` 使 `Jest` 在后台运行。`test-watch` 由 `npm run test-watch` 启动(只有 `test` 和 `start` 不需要 `run`)。只需要运行一次 `test-watch`，`Jest`(在监视模式下)会注意到任何源代码更改并重新运行测试。以下是输出示例：

```

PASS  __tests__/_password.test.js
PASS  __tests__/_generate-password.test.js

```

```

Test Suites: 2 passed, 2 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.502s
Ran all test suites.

```

#### Watch Usage

- > Press o to only run tests related to changed files.
- > Press p to filter by a filename regex pattern.
- > Press t to filter by a test name regex pattern.
- > Press q to quit watch mode.
- > Press Enter to trigger a test run.

到目前为止，已经定义了项目依赖关系。接下来，需要设置 `Webpack` 构建过程，以便可以将 `JSX` 转换为 `JS`。为此，请使用代码清单 19.2 在根目录中创建 `webpack.config.js` 文件(`ch16/password/webpack.config.js`)。



## 代码清单19.2 Webpack配置

```

module.exports = {
  entry: './jsx/app.jsx',
  output: {
    path: __dirname + '/dist/',
    filename: 'bundle.js'
  },
  devtool: '#sourcemap',
  stats: {
    colors: true,
    reasons: true
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
}

```

设置项目入口  
(可以有多个入口)

设置Source Map以便可以在DevTools  
中查看正确的源代码行号

应用Babel, 并使用  
package.json中的Babel配置

现在可以在 webpack.config.js 中通过定义配置来构建项目。入口点是 jsx 文件夹中的 app.jsx 文件，目标文件夹是 dist。此外，配置还设置了 Source Map 和 babel-loader(将 JSX 转换为 JS)。

使用 ./node\_modules/.bin/webpack 或 ./node\_modules/.bin/webpack -w(监视文件更改)执行构建。使用 -w(监视模式)时，每次文件更改 Webpack 都会重新执行构建。也就是说，每次在 Notepad(我不喜欢使用 IDE)中单击 Save 按钮时就会执行构建。监视模式在开发阶段非常有用！

可以通过使用 --config 命令指定不同的文件名来创建多个 webpack.config.js:

```
$. /node_modules/.bin/webpack --config production.config.js
```

为了方便，可以在 package.json 中为每一个配置文件添加新的 npm 脚本。

最重要的一点是：使用 Webpack 很简单，也很有趣，因为它默认支持 CommonJS/Node 模块。不需要 Browserify 或任何其他模块加载器。有了 Webpack，就像为浏览器 JavaScript 编写 Node 程序一样！

## 19.2 HTML 主页

接下来创建 index.html 文件，该文件中有一个 id 为 password 的容器，并且引入了文件 js/bundle.js，如代码清单 19.3 所示(ch16/password/index.html)。

## 代码清单19.3 主机HTML文件

```

<!DOCTYPE html>
<html>

  <head>
    <link href="css/bootstrap.css" rel="stylesheet" type="text/css"/>
  </head>

```

```

<body class="container">
  <h1>Password Input Widget</h1>
  <div id="password"></div>
  <script src="dist/bundle.js" ></script>
</body>
</html>

```

← 加载打包后的脚本文件

现在应该已经设置好并开始开发。在开发过程中以增量方式进行测试是一种不错的方式，这样用于寻找错误的区域会尽可能小。接下来执行一项快速测试以查看设置是否正常工作，就像在第 18 章中所做的那样。执行以下操作：

- ① 安装所有依赖(npm install)。只需要执行一次。
- ② 将 `console.log('Painless JavaScript password!')` 写入 `jsx/app.jsx` 文件。
- ③ 使用 `npm start` 运行应用。之后应用会保持运行，`-w` 会在代码更改时自动重新构建文件。
- ④ 从项目根目录启动本地 Web 服务器。
- ⑤ 在浏览器中打开链接 `http://localhost:8080`。
- ⑥ 打开浏览器控制台(如 Chrome DevTools)。你应该看到消息“Painless JavaScript password!”。

## 19.3 实现强密码模块

强密码模块是存放在 `password/js` 文件夹中的 `generate-password.js` 文件。用于该文件的测试在 `password/__tests__/generate-password.test.js` 中。这个模块在调用时会返回随机密码，该密码是包含如下不同类型字符的组合：

- 特殊字符：!`@#%&*()_+{}:"<>?\\'./~`
- 小写字母：`abcdefghijklmnopqrstuvwxyz`
- 大写字母：`ABCDEFGHIJKLMNOPQRSTUVWXYZ`
- 数字：`0123456789`

这些类别以及长度和随机性将确保密码足够安全。使用 TDD/BDD，我们应该首先实现测试。

### 19.3.1 测试

从 `generate-password.test.js` 中的测试开始。请将测试文件存储在 `__tests__` 文件夹中，以便 Jest 可以找到它们，如代码清单 19.4 所示(`ch16/password/__tests__/generate-password.test.js`)。

代码清单 19.4 密码模块的测试

```

const generatePassword = require('../js/generate-password.js')
const pattern = /^[A-Za-z0-9!\@#\$\%\^&\*\(\)\_\+\{\}\|\"'<>?\\'./~\
  [\]\]/^\\.\.\`~}{8,16}$/

describe('method generatePassword', ()=>{
  let password, password2

```

← 为符合所有标准的密码定义 RegEx 模式

```

it('returns a generated password of the set pattern', () => {
  password = generatePassword()
  expect(password).toMatch(pattern)
})

it('return a new value different from the previous one', () => {
  password2 = generatePassword()
  expect(password2).toMatch(pattern)
  expect(password2).not.toEqual(password)
})

```

测试生成的密码是否匹配模式

测试该方法是否返回一个新的密码

首先声明 `password` 变量并导入 `generate-password.js`。正则表达式检查密码的内容和长度，但这并不完美，因为还没有检查每个密码是否至少有一个这样的字符，不过就目前来说，可以正常工作：

```

let password,
    password2,
    pattern = /^[A-Za-z0-9!@#%$%^&*()\_+{}|\:~<>?~\|
    \[\]\\/'\.,\.\~\~]{8,16}$/

```

在测试套件中添加 `describe('method generatePassword')`。这就是需要测试的内容，其中 `generatePassword` 是从 `generate-password.js` 模块中导出的函数。

如第 16 章所述，通过 BDD 风格的 `expect` 语句实现进行单元测试的测试套件 `it`。至少要检查密码的正则表达式模式：

```

describe('method generatePassword', () => {
  it('returns a generated password of the set pattern', () => {
    password = generatePassword()
    expect(password).toMatch(pattern)
  })
  it('returns a new value different from the previous one', () => {
    password2 = generatePassword()
    expect(password2).not.toEqual(password)
  })
})

```

如果在每次调用 `generatePassword()` 时生成的密码都一样，怎么办？如果在 `generatepassword.js` 中密码是硬编码的，怎么办？那会非常糟糕！所以，对第二个测试套件的期望是：第二次生成的密码和上一次生成的密码是不同的。

### 19.3.2 代码

在 `js/generate-password.js` 中实现一个强密码模块，以便可以立即对它进行 TDD/BDD。也就是说，先写测试，再写代码。如下是一个通用的密码生成器，它使用三组字符来满足强密码规则：

```

const SPECIALS = '!@#$%^&*()_+{}:"<>?\\|[]\`',./~'
const LOWERCASE = 'abcdefghijklmnopqrstuvwxyz'
const UPPERCASE = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
const NUMBERS = '0123456789'

```



```

const ALL = `${SPECIALS}${LOWERCASE}${UPPERCASE}${NUMBERS}`

const getIterable = (length) => Array.from({length},
  (_, index) => index + 1)

const pick = (set, min, max) => {
  let length = min
  if (typeof max !== 'undefined') {
    length += Math.floor(Math.random() * (max - min))
  }
  return getIterable(length).map(() => (
    set.charAt(Math.floor(Math.random() * set.length))
  )).join('')
}

const shuffle = (set) => {
  let array = set.split('')
  let length = array.length
  let iterable = getIterable(length).reverse()
  let shuffled = iterable.reduce((acc, value, index) => {
    let randomIndex = Math.floor(Math.random() * value)
    [acc[value - 1], acc[randomIndex]] = [acc[randomIndex], acc[value - 1]]
    return acc
  }, [...array])
  return shuffled.join('')
}

module.exports = () => {
  let password = (pick(SPECIALS, 1)
    + pick(LOWERCASE, 1)
    + pick(NUMBERS, 1)
    + pick(UPPERCASE, 1)
    + pick(ALL, 4, 12))
  return shuffle(password)
}

```

通过+1来避免将0作为值，并使用隐式返回

定义pick()函数，它从min到max之间的集合返回字符

用空字符串创建一个可迭代的元素

随机打乱字符来获得随机性

反转迭代，从max到min的集合中获取值

使用reducer获取随机数组

定义满足密码小部件的规则

导出的函数(分配给 `module.exports`)调用 `shuffle()` 方法，`shuffle()` 随机地在字符串中移动字符。`shuffle()` 获取由 `pick()` 生成的密码，`pick()` 使用多组字符来确保生成的密码至少包括一个某一组中的字符(数字、大写字母、特殊字符等)。密码的最后一部分由联合字符集 `ALL` 中的更多随机元素组成。

在项目根目录(password 文件夹)下执行命令 `jest __tests__/generate-password.test.js` 或 `npm test __tests__/generate-password.test.js`，运行用于 `password/__tests__/generate-password.js` 的单元测试。应该会输出类似于以下内容的消息：

```

jest  __tests__/generate-password.test.js

PASS  __tests__/generate-password.test.js
  method generatePassword
    ✓ returns a generated password of the set pattern (4ms)
    ✓ return a new value different from the previous one (2ms)

Test Suites: 1 passed, 1 total
Tests:      2 passed, 2 total
Snapshots:  0 total
Time:       1.14s
Ran all test suites matching "__tests__/generate-password.test.js".

```

## 19.4 实现 Password 组件

要做的下一件合乎逻辑的事情是处理主组件 Password。根据 TDD 模式，必须再次从测试开始：在本例中是 UI 测试，因为需要测试像单击这样的行为。

### 19.4.1 测试

创建名为 `__tests__/password.test.js` 的 UI 测试文件。我们已经在第 16 章中介绍了这个文件，所以在这里给出带有一些注释的完整示例，如代码清单 19.5 所示(ch16/password/ `__tests__/password.test.js`)。

代码清单19.5 Password组件的规范

```
describe('Password', function() {
  it('changes after clicking the Generate button', (done)=>{
    const TestUtils = require('react-addons-test-utils')
    const React = require('react')
    const ReactDOM = require('react-dom')
    const Password = require('../jsx/password.jsx')
    const PasswordGenerate = require('../jsx/password-generate.jsx')
    const PasswordInfo = require('../jsx/password-info.jsx')
    const PasswordInput = require('../jsx/password-input.jsx')
    const PasswordVisibility = require('../jsx/password-visibility.jsx')

    const fd = ReactDOM.findDOMNode

    let password = TestUtils.renderIntoDocument(<Password
      upperCase={true}
      lowerCase={true}
      special={true}
      number={true}
      over6={true}
    />)

    let rules = TestUtils.scrRenderedDOMComponentsWithTag(password,
      'li')
    expect(rules.length).toBe(5)

    expect(fd(rules[0]).textContent).toEqual('Must have
      ➡ at least one upper-case character')
    let generateButton = TestUtils.findRenderedDOMComponentWithClass(password,
      ➡ 'generate-btn')
    expect(fd(rules[1]).firstChild.nodeName.toLowerCase()).
      ➡ toBe('#text')
    TestUtils.Simulate.click(fd(generateButton))
    expect(fd(rules[1]).firstChild.nodeName.toLowerCase()).
      ➡ toBe('strike')
    done()
  })
})
```

导入相关库

创建一个React组件，JSX语法由babel-jest包(Jest的一部分，参见<https://github.com/facebook/jest/tree/master/packages/babel-jest>)支持

获取<li>元素

获取密码生成按钮

检查第一个<li>的文本是否与期望值匹配

单击Generate按钮

满足第二条规则

不满足第二条规则，即子节点是文本而不是<strike>标签

可以扩展此测试用例以检查所有属性和规则是否通过,将这个扩展作为课后作业(在本章末尾的“测验”部分查看更多信息)。在另一个测试套件中,提供不同的属性组合,然后进行测试是不错的主意。

现在,执行测试(`npm test` 或 `jest`)会抛出如下错误:

```
Error:Cannot find module '../jsx/password.jsx'from'password.test.js'
```

这是测试驱动开发的正常现象,原因在于我们先写测试。你现在需要做的主要事情就是实现 Password 组件。

## 19.4.2 代码

接下来,创建带有初始状态的 Password 组件。状态变量如下:

- **strength**: 表示当前密码强度的对象(一组规则,每条规则被设置为真或假,真假取决于是否满足标准)。
- **password**: 当前密码。
- **visible**: 密码输入字段是否可见。
- **ok**: 密码是否满足所有规则,是否允许用户保存(启用 Save 按钮)。

想象一下,在实现这个组件之后,来自另一支团队的开发者想要使用该组件,但要使用较严格的密码标准。最好的方法是将密码规则的代码抽象到一个单独的文件中。在使用 `password.jsx` 之前将规则抽象出来,创建一个名为 `rules.js`(`ch16/password/js/rules.js`)的文件,参见代码清单 19.6。这个文件实现了密码规则,可以在 `password.jsx` 中使用这些规则来执行验证并显示警告消息。保持规则的独立可以使得在将来更改、添加或删除规则时变得更加简单明了。

代码清单19.6 密码强度规则

```
module.exports = {
  upperCase: {
    message: 'Must have at least one upper-case character',
    pattern: /[A-Z]+/
  },
  lowerCase: {
    message: 'Must have at least one lower-case character',
    pattern: /[a-z]+/
  },
  special: {
    message: 'Must have at least one special character (#$@!&%...)',
    pattern: /[!@#$%^&*()\_+{}|:~<>?\\[\]\/\.\.\~]+/
  },
  number: {
    message: 'Must have at least one number',
    pattern: /[0-9]+/
  },
  'over6': {
    message: 'Must be more than 6 characters',
    pattern: /(.{6,})/
  }
}
```



基本上你有一堆规则，每个规则对象都使用以下结构：

- 键，如 `over6`。
- 信息，如 “Must be more than 6 characters”。
- 正则表达式，如 `(/{6,})/`。

在 `password.jsx` 中，执行以下步骤：

- ① 使用 `upperCase`、`lowerCase`、`special`、`number` 和 `over6` 规则进行渲染。
- ② 检查规则是否已被渲染(长度为 5)。
- ③ 规则 1 不满足。
- ④ 单击 `Generate` 按钮。
- ⑤ 规则 2 满足。

让我们来实现这个组件。导入依赖，传递初始状态构建组件，如代码清单 19.7 所示 (`ch6/password/jsx/password.jsx`)。

代码清单 19.7 实现 Password 组件

```
const React = require('react')
const ReactDOM = require('react-dom')
const generatePassword = require('../js/generate-password.js')
const rules = require('../js/rules.js')
const PasswordGenerate = require('../password-generate.jsx')
const PasswordInfo = require('../password-info.jsx')
const PasswordInput = require('../password-input.jsx')
const PasswordVisibility = require('../password-visibility.jsx')

class Password extends React.Component {
  constructor(props) {
    super(props)
    this.state = {strength: {}, password: '', visible: false, ok: false}
    this.generate = this.generate.bind(this)
    this.checkStrength = this.checkStrength.bind(this)
    this.toggleVisibility = this.toggleVisibility.bind(this)
  }
  ...
}
```

接下来是实现检查密码强度的方法：

```
checkStrength(event) {
  let password = event.target.value
  this.setState({password: password})
  let strength = {}
```

以下代码段将遍历每个属性(`upperCase`、`over6` 等)，并使用规则中的正则表达式检查当前密码。如果满足规则，就将 `strength` 对象中的这个属性设置为 `true`：

```
Object.keys(this.props).forEach((key, index, list)=>{
  if (this.props[key] && rules[key].pattern.test(password)) {
    strength[key] = true
  }
})
```

由于 `this.setState()` 是异步的，因此使用回调来提供依赖更新后状态的逻辑。在本例的回调中，检查 `strength` 对象(`this.state.strength`)中的属性数量是否等于规则数量(`props`)。这只是初步检查，在循环中检查每个属性是一种更健壮的解决方案，但目前代码可以正常工作。如果数量匹配(即满足密码强度的所有规则)，就将 `ok` 设置为 `true`：

```
this.setState({strength: strength}, ()=>{
  if (Object.keys(this.state.strength).length ==
    Object.keys(this.props).length) {
    this.setState({ok: true})
  } else {
    this.setState({ok: false})
  }
})
```

下一个方法用于切换密码输入框的显隐状态。当生成新密码时这是一项有用的功能，因为可能需要保存密码(或需要帮助记住密码)：

```
toggleVisibility() {
  this.setState({visible: !this.state.visible}, ()=>{
  })
}
```

接下来是 `generate()` 方法，这个方法使用 `js/generate-password.js` 模块创建随机密码。将 `visible` 设置为 `true`，确保用户可以看到新生成的密码，密码生成之后，调用 `checkStrength()` 来检查密码强度。通常情况下，条件都能得到满足，用户也能够通过单击 `Save` 按钮进行保存：

```
generate() {
  this.setState({visible: true, password: generatePassword()}, ()=>{
    this.checkStrength({target: {value: this.state.password}})
  })
}
```

在 `render()` 函数中，`Password` 处理规则并渲染其他一些 `React` 组件：

- `PasswordInput`：密码输入框(input)
- `PasswordVisibility`：切换密码显隐状态(checkbox 类型的 input)
- `PasswordInfo`：密码强度规则列表(ul)
- `PasswordGenerate`：密码生成按钮(button)

首先处理规则并确定满足哪些规则(`isCompleted`)。使用箭头函数 `() => {}` 代替使用 `_this` 或 `bind(this)` 模式来传递上下文。这些方式没有什么大的区别，可以任选一种方法使用。

`Object.keys` 将哈希表压缩成一个数组，数组成员是对象的属性的键名。使用 `map()` 迭代这个数组，并使用具有 `key`、`rule` 和 `isCompleted` 属性的对象构造一个新的数组：

```
render() {
  var processedRules = Object.keys(this.props).map((key)=>{
    if (this.props[key]) {
      return {
        key: key,
        rule: rules[key],
        isCompleted: this.state.strength[key] || false
      }
    }
  })
  // return ...
}
```

## 实现 Password 组件的 render() 方法

一旦处理好的规则数组准备就绪, 就可以开始渲染组件。请记住, 在 JavaScript 中 class 是一个特殊的单词, 这就是为什么使用 className 而不是 class 的原因, 参见代码清单 19.8(ch16/password/jsx/ password.jsx)。

代码清单 19.8 实现 render() 方法

```
return (
  <div className="well form-group col-md-6">
    <label>Password</label>
    <PasswordInput
      name="password"
      onChange={this.checkStrength}
      value={this.state.password}
      visible={this.state.visible}/>
    <PasswordVisibility
      checked={this.state.visible}
      onChange={this.toggleVisibility}/>
    <PasswordInfo rules={processedRules}/>
    <PasswordGenerate onClick={this.generate}>
      Generate
    </PasswordGenerate>
    <button className={'btn btn-primary' +
      ((this.state.ok)? '' : ' disabled')}>
      Save
    </button>
  </div>
)
```

每次输入框中的内容发生变化时校验密码强度

复选框发生变化时切换密码显隐状态

单击按钮时生成新的密码

让我们来更详细地介绍最重要的部分。PasswordInput 是受控组件(有关受控组件和不受控组件之间的详细比较, 请参见第 5 章)。使用 this.checkStrength 回调来监听每次变化, 回调中使用 e.target.value 获取最新的值, 所以不需要 ref:

```
<PasswordInput name="password" onChange={this.checkStrength}
  value={this.state.password} visible={this.state.visible}/>
```

与 PasswordInput 类似, PasswordVisibility 也是受控组件, 用于变更的事件处理程序是 this.toggleVisibility:

```
<PasswordVisibility checked={this.state.visible}
  onChange={this.toggleVisibility}/>
```

将 processedRules 对象传递给规则列表, PasswordGenerate 按钮会触发 this.generate:

```
<PasswordInfo rules={processedRules}/>
<PasswordGenerate onClick={this.generate}>Generate</PasswordGenerate>
```

Save 按钮是否可用基于 this.state.ok 的值。不要忘记 disabled 前面的空格, 否则将获得的是 btn-primarydisabled 类而不是 btn-primary 和 disabled 这两个类:

```
<button className={'btn btn-primary' +
  ((this.state.ok)? '' : ' disabled')}>Save</button>
</div>
))
```

其他组件是木偶组件, 它们只是渲染类和传递属性, 如代码清单 19.9(ch16/password/



jsx/password-generate.jsx)、19.10(ch16/password/jsx/password-input.jsx)和 19.11(ch16/password/jsx/password-visibility.jsx)所示。

代码清单19.9 PasswordGenerate组件

```
const React = require('react')
class PasswordGenerate extends React.Component {
  render() {
    return (
      <button {...this.props} className="btn generate-btn">
        {this.props.children}</button>
      )
    )
  }
}
module.exports = PasswordGenerate
```

代码清单19.10 PasswordInput组件

```
const React = require('react')
class PasswordInput extends React.Component {
  render() {
    return (
      <input className="form-control"
        type={this.props.visible ? 'text' : 'password'}
        name={this.props.name}
        value={this.props.value}
        onChange={this.props.onChange}/>
    )
  }
}
module.exports = PasswordInput
```

代码清单19.11 PasswordVisibility组件

```
const React = require('react')
class PasswordVisibility extends React.Component {
  render() {
    return (
      <label className="form-control">
        <input className=""
          type="checkbox"
          checked={this.props.checked}
          onChange={this.props.onChange}/> Show password
        </label>
      )
    )
  }
}
module.exports = PasswordVisibility
```

使用属性值  
控制组件  
是否选中

通过属性  
在父级触  
发事件

让我们来看一下 PasswordInfo 组件，参见代码清单 19.12(ch16/password/jsx/password-info.jsx)。这个组件接收经过处理的规则数组并遍历这个数组。如果 isCompleted 为 true，就将<strike>添加到<li>中。<strike>是一个 HTML 标签，它将一条删除线应用在文本上。这也是需要在 password.test.js 中测试的内容。

代码清单19.12 PasswordInfo组件

```

const React = require('react')
class PasswordInfo extends React.Component {
  render() {
    return (
      <div>
        <h4>Password Strength</h4>
        <ul>
          {this.props.rules.map(function(processedRule, index, list){
            if (processedRule.isCompleted)
              return <li key={processedRule.key}>
                <strike>{processedRule.rule.message}</strike>
              </li>
            else
              return <li key={processedRule.key}>
                {processedRule.rule.message}</li>
          })}
        </ul>
      </div>
    )
  }
}

module.exports = PasswordInfo

```

通过属性检查规则是否满足

通过属性使用 rules.js 中提供的消息内容

到此，我们完成了 password.jsx 文件！现在可以重新运行测试了，但是不要忘记使用 `npm run build` 或 `npm run build-watch` 重新编译。如果跟随我们的示例进行操作，运行 `npm test` 之后，应该会看到下面这样的内容：

```

Using Jest CLI v0.5.10
PASS __tests__/generate-password.test.js (0.03s)
PASS __tests__/password.test.js (1.367s)
2 tests passed (2 total)
Run time: 2.687s

```

## 19.5 实践

要查看密码小部件在实践中的应用，还需要完成一个小步骤：创建 `jsx/app.jsx`，它是组件的示例文件。以下是在应用中渲染 Password 组件的方法：

```

const React = require('react')
const ReactDOM = require('react-dom')
const Password = require('./password.jsx')
ReactDOM.render(<Password
  upperCase={true}
  lowerCase={true}
  special={true}
  number={true}
  over6={true}/>,
  document.getElementById('password'))

```

可以像任何其他前端应用一样运行文件。我个人比较喜欢 `node-static`(<https://github.com/cloudhead/node-static>), 或者在 <http://reactquickly.co/demos> 上查看在线示例。请注意, 当所有规则都满足时, `Save` 按钮是如何变为可用状态的, 如图 19.3 所示。

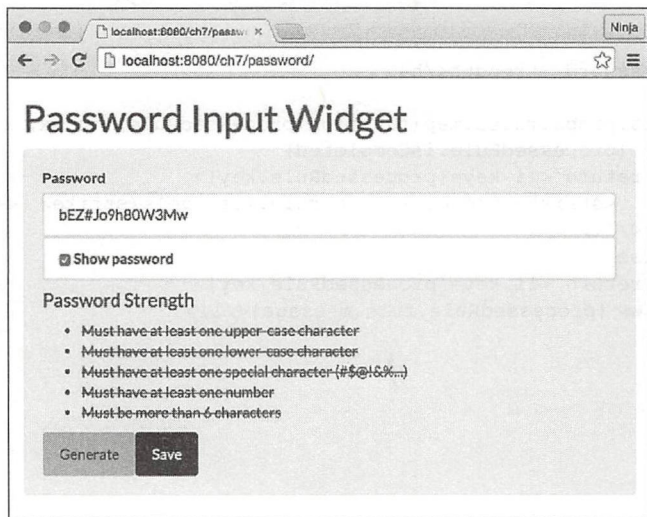


图 19.3 所有规则都满足时 `Save` 按钮变成可用状态

### CI 和 CD

最好的软件工程实践会持续地在本地编写和运行测试。当与部署过程和自动化相结合时, 测试会更有价值。称这些过程为持续集成(Continuous Integration, CI)和持续部署(Continuous Deployment, CD), 它们对于加速和自动化软件交付非常重要。

强烈建议你为任何项目设置 CI/CD, 而不仅仅是把它们当做原型。有很多好的软件即服务(Software-as-a-Service, SaaS)和托管解决方案。使用它们为本项目中的测试设置 CI/CD 环境不会花费太长时间。例如, 使用 AWS、Travis CI 或 CircleCI, 需要做的就是根据运行的环境来配置项目, 然后提供诸如 `npm test` 这样的测试命令。甚至可以将这些 SaaS CI 与 GitHub 集成在一起, 这样你和你的团队就可以在 GitHub 请求上看到 CI 消息(通过、失败、有多少次失败以及在哪里失败)。

Amazon Web Services 提供了自己的托管服务: CodeDeploy、CodePipeline 和 CodeBuild。有关 AWS 服务的更多信息, 请参阅 Node University: <https://node.university/p/aws-intermediate>。如果更喜欢自己搭建解决方案而不是托管解决方案, 可以查看 Jenkins(<https://jenkins.io>)和 Drone(<https://github.com/drone/drone>)。

## 19.6 测验

为了有更多收获, 不妨尝试下面的操作:

- 测试你能想到的任何情形: 例如, 只输入小写字母(如 `r`), 并看到小写规则处于满足状态, 但其他规则并不满足。



- 使用云 SaaS CI 提供商(AWS、TravisCI、CircleCI 等)注册一个免费账户，并设置项目，使其在云 CI 环境中运行。

将代码提交到本书的 GitHub 仓库(<https://github.com/azat-co/react-quickly/>)中 ch16 目录下的新文件夹中。

## 19.7 小结

- 按照规范，Jest 测试文件存储在 \_\_tests\_\_ 文件夹中。
- 可以使用 react-dom/test-utils 或 react-test-renderer/shallow 进行常规或浅渲染。
- Jest(v19)测试可以使用 JSX 编写，原因是 Jest 会自动转换 JSX。
- 要启用自动重新运行测试(推荐在开发环境中使用)，请使用 `jest --watch`。

# 第 20 章

## 使用 Jest、Express 和 MongoDB 实现自动完成

本章内容:

- 项目结构和 Webpack 配置
- 实现 Web 服务器
- 添加浏览器脚本
- 创建服务器模板
- 实现自动完成组件

这个项目的首要目标是将你在本书中学到的许多技术结合起来,例如组件构成、状态、表单元素、测试、如何通过 API 服务器获取数据、存储数据、如何实现一台简单的 Express 服务器以及使用同构 React 进行渲染。虽然已经完成了书中大部分的事情,但复习还是很有必要的——特别是间歇性地复习!

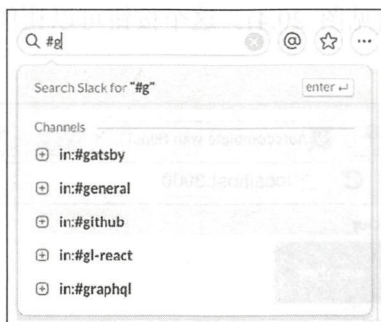


图 20.1 在 Slack 应用中输入内容时,小部件提供相关匹配项

在本章中，将构建一个全面的组件，并为其提供后端服务。这个小项目非常接近你在现实工作中最有可能遇到的真实项目。

简单地说，这个项目将指导你构建一个自动完成组件，这个组件在视觉和功能上都类似于 Slack(流行的消息应用)和 Google(流行的搜索引擎)的自动完成组件，如图 20.1 所示。为了简单起见，该小部件将与聊天应用程序中的房间名称一起使用。

自动完成小部件(如图 20.2 所示)具有以下内容：

- 输入框：常态显示，但初始值为空。
- 根据输入内容过滤出的选择列表：至少匹配到一项时显示。
- Add 按钮：当没有任何匹配时显示。

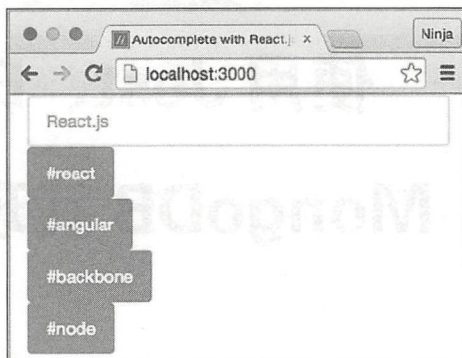


图 20.2 输入框中的内容为空时的自动完成表单

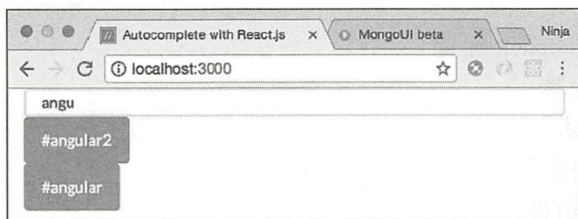


图 20.3 输入 angu 过滤匹配时仅显示 angular 和 angular2

房间名称使用输入的字符作为每个选项的第一个字符进行过滤。通过作简单比较，自动填充房间的名称(见图 20.3)。例如，如果有名为 angular、angular2 和 react 的房间，并且输入 angu，将只显示匹配的 angular 和 angular2，而不显示 react。

如果没有匹配，怎么办？可以使用 Add 按钮添加新的选项。为了使用方便，Add 按钮只有在没有任何匹配时才显示(见图 20.4)。这个按钮可以让你将新的输入永久保存在数据库中。

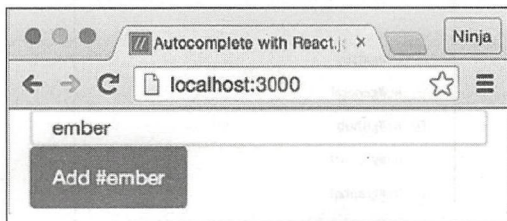


图 20.4 当没有匹配时显示 Add 按钮



新选项通过对 REST API 的 XHR 调用被保存到数据库中。可以在将来的匹配中使用这个新添加的房间名称(见图 20.5), 就像初始的房间名称列表一样。

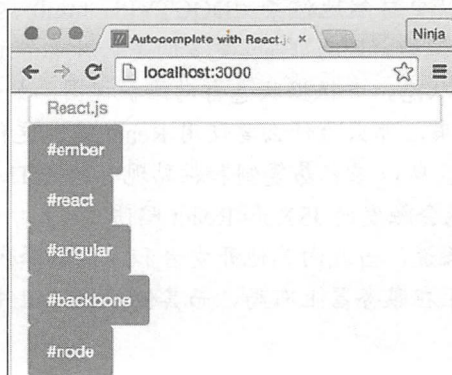


图 20.5 房间名称已保存且现在出现在列表中

为了实现自动完成小部件, 需要完成如下步骤:

- ① 安装依赖。
- ② 使用 Webpack 设置构建过程。
- ③ 使用 Jest 编写测试用例
- ④ 实现一台连接到 MongoDB 的 Express REST API 服务器, 并且作为小部件的静态服务器。
- ⑤ 实现 Autocomplete React 组件
- ⑥ 使用 Autocomplete 和 Handlebars 实现示例。

你将在服务器上渲染 React 组件, 使用 Jest 进行测试, 并使用 axios 进行 AJAX/XHR 请求。

让我们从设置项目开始。

## 20.1 项目结构和 Webpack 配置

为了让你对技术栈有一个大致的了解, 下面列出了在这个项目中将使用的技术和库:

- Node.js 和 npm 用于编译 JSX 和下载依赖, 如 React。
- Webpack 作为构建工具。
- Jest 作为测试引擎。
- Express 作为 Web 服务器, 使用本地 MongoDB Node.js 驱动程序访问 MongoDB, 从而保存自动完成选项。
- 使用 Handlebars 模板引擎进行布局。

为什么使用 Handlebars 而不是 React?

我更喜欢使用 Handlebars 进行布局, 有以下几个原因:

首先, React 难以输出未转义的 HTML, 而使用一种怪异的语法进行输出。但是, 输出未转义的 HTML 也是需要为同构 React 和服务器端渲染所做的工作。虽然未转义的

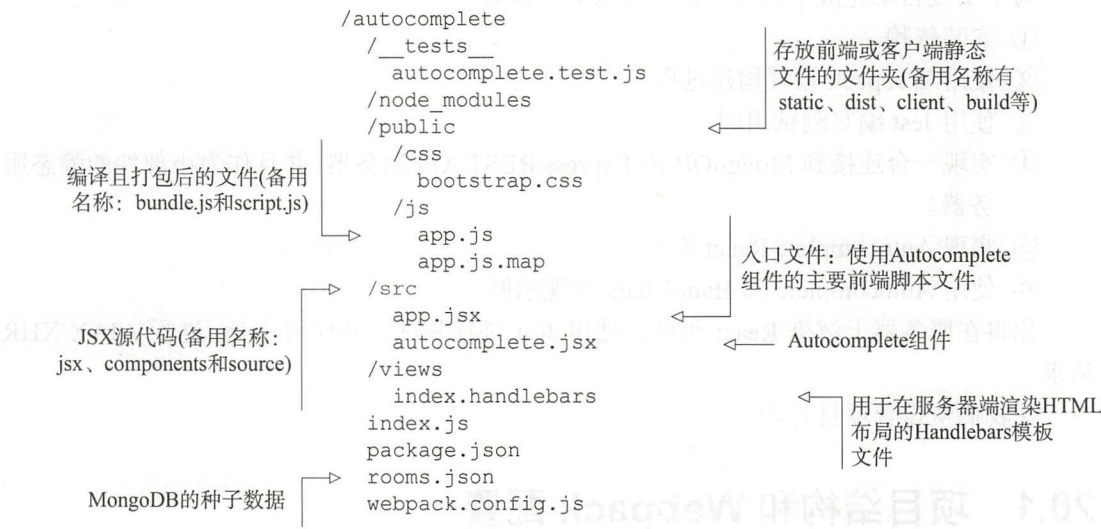
HTML 会使应用受到跨站脚本攻击<sup>1</sup>的危险，但是在服务器上进行渲染可以控制 HTML 字符串。

其次，Handlebars 可以更自然地渲染<!DOCTYPE html>这样的内容。React 不能像 Handlebars 那样自然地完成渲染，因为 React 更多的是针对单个元素而不是整个页面。

再次，React 用于管理状态，并根据状态自动维护视图。如果正在做的事情是从 React 组件呈现静态 HTML 字符串，那么为什么要使用 React 呢？这种情况下 React 是多余的。Handlebars 类似于 HTML，所以很容易复制和粘贴现有的 HTML 代码，而不必考虑在将 HTML 转换为 React 时可能会触发的 JSX 和 React 陷阱。

最后，就我个人经验来说，当我向其他开发者和学生解释代码功能时，一些人很难理解一部分 React 组件仅用于在服务器上布局，而其他 React 组件既用于客户端也用于服务器端的情况。

附录 A 涵盖了这些工具的安装方法，这里不再赘述。继续创建一个新的名为 autocomplete 的项目文件夹。文件夹结构如下：



`__tests__` 文件夹用于 Jest 测试。现在你应该很熟悉: `node_modules` 文件夹用于存放 Node.js 依赖(来自 npm 的 `package.json`)。 `public`、 `public/css` 和 `public/js` 文件夹包含应用的静态文件。

**命名**

命名对于良好的软件工程是至关重要的，因为好的名称能够提供关键的信息。通过名称可以知道很多关于脚本、文件、模块或组件的信息，而不必阅读源代码、测试或文档(可能不存在！)。

就像你已经熟悉将 JSX 文件放入 `jsx` 文件夹，并使用 `build` 作为编译文件的目标文件夹，

1 跨站脚本(Cross-Site Scripting, XSS)攻击的特点是：攻击者将恶意代码注入用户信任但包含 XSS 漏洞的合法网站。例如，攻击者可以在一个转发文本的论坛上发布一些包含<script>元素的错误代码。论坛上的所有访客将执行恶意代码。有关 XSS 的更多信息，请参阅 Jakob Kallin 和 Irene Lobo Valbuena 的文章 “Excess XSS: A Comprehensive Tutorial on Cross-Site Scripting”，<https://excess-xss.com>



不过我已经开始使用其他名称了。这是因为，你会遇到许多不同的约定。每个项目可能会有不同的结构，结构的变化可大可小。作为一名开发者，需要熟悉配置诸如 Webpack 之类的工具和 Express 等库来处理任何名称。出于这个原因，也为了增加多样性，在本章中，我使用 public 代替 build(另外，public 也是由 Express 提供的静态文件的约定)，使用 src 代替 jsx(可能还有其他源文件，而不仅仅是 JSX)，等等。

Webpack 根据依赖关系和 JSX 源代码 src/app.jsx 构建打包出 public/js/app.js 文件。Autocomplete 组件的源代码位于 src/autocomplete.jsx 文件中。

views 文件夹用于 Handlebars 模板。如果对自己的 React 技术有信心，可以不使用模板引擎；你可以使用 React 作为 Node.js 模板引擎！

在项目根目录中包含如下文件：

- webpack.config.js: 启用构建任务
- package.json: 包含项目元数据
- rooms.json: 包含 MongoDB 种子数据
- index.js: 使用 Express 服务器及其 API 服务器的路由(GET 和 POST /rooms)

不要忘记，为了避免手动安装每个依赖的确切版本，可以将 package.json 文件从代码清单 20.1 复制到根文件夹中(ch20/autocomplete/package.json)，然后运行 npm install。

代码清单20.1 项目的依赖和设置

```
{
  "name": "autocomplete",
  "version": "1.0.0",
  "description": "React.js autocomplete component with Express.js, and
  ➡ MongoDB example.",
  "main": "index.js",
  "scripts": {
    "test": "jest",
    "start": "npm run build && ./node_modules/.bin/node-dev index.js",
    "build": "./node_modules/.bin/webpack",
    "seed": "mongoimport rooms.json --jsonArray --collection=rooms
    ➡ --db=autocomplete"
  },
  "keywords": [
    "react.js",
    "express.js",
    "mongodb"
  ],
  "author": "Azat Mardan",
  "license": "MIT",
  "babel": {
    "presets": [
      "react"
    ]
  },
  "dependencies": {
    "babel-register": "6.11.6",
    "body-parser": "1.13.2",
    "express": "4.13.3",
    "mongoose": "4.11.8",
    "morgan": "1.7.0",
    "react": "15.0.1",
    "react-dom": "15.0.1",
    "react-jss": "4.0.0",
    "react-router": "2.7.0",
    "react-router-dom": "1.1.1",
    "redux": "3.6.0",
    "redux-thunk": "2.1.0",
    "webpack": "1.12.2"
  }
}
```

在服务器端导入和转换 JSX



```

    "body-parser": "1.13.2",
    "compression": "1.5.1",
    "errorhandler": "1.4.1",
    "express": "4.13.1",
    "express-handlebars": "2.0.1",
    "express-validator": "2.13.0",
    "mongodb": "2.0.36",
    "morgan": "1.6.1"
  },
  "devDependencies": {
    "axios": "0.13.1",
    "babel-core": "6.10.4",
    "babel-loader": "6.2.4",
    "babel-preset-react": "6.5.0",
    "jest-cli": "13.2.3",
    "node-dev": "3.1.3",
    "react": "15.5.4",
    "react-dom": "15.5.4",
    "webpack": "1.13.1"
  }
}

```

用于记录HTTP请求的Express插件(中间件) →

← 服务器端Web框架Express

← 连接到MongoDB数据库的库

当然，如果想有一个能正常工作的应用，建议使用与本书相同的版本。另外，不要忘记使用 `npm i` 安装 `package.json` 中的依赖。

`package.json` 中的 `scripts` 部分比较有趣：

```

"scripts": {
  "test": "jest",
  "start": "./node_modules/.bin/node-dev index.js",
  "build": "./node_modules/.bin/webpack",
  "seed": "mongoimport rooms.json --jsonArray --collection=rooms
  ➡ --db=autocomplete"
},

```

`test` 用来运行测试用例，`start` 用来编译及启动服务器。还可以使用 `$ npm run seed` 为房间名称添加种子数据。数据库名称是 `autocomplete`，集合名称是 `rooms`。如下是 `rooms.json` 文件的内容：

```

[ { "name": "react",
  { "name": "node",
  { "name": "angular",
  { "name": "backbone"}]

```

当运行 `seed` 命令时，会输出如下信息(MongoDB 必须作为单独的进程运行)：

```

> autocomplete@1.0.0 seed /Users/azat/Documents/Code/
➡ react-quickly/ch20/autocomplete
> mongoimport rooms.json --jsonArray --collection=rooms --db=autocomplete

2027-07-10T07:06:28.441-0700    connected to: localhost
2027-07-10T07:06:28.443-0700    imported 4 documents

```

至此已经定义了项目依赖，现在需要设置 Webpack 构建过程，以便可以使用 ES6 和转换 JSX。为此，请使用代码清单 20.2(ch20/autocomplete/webpack.config.js)在根目录中创建 `webpack.config.js` 文件。

代码清单20.2 Webpack配置

```
module.exports = {
  entry: './src/app.jsx',
  output: {
    path: __dirname + '/public/js/',
    filename: 'app.js'
  },
  devtool: '#sourcemap',
  stats: {
    colors: true,
    reasons: true
  },
  module: {
    loaders: [
      {
        test: /\.jsx?$/,
        exclude: /(node_modules)/,
        loader: 'babel-loader'
      }
    ]
  }
}
```

设置项目入口文件  
(可以有多个入口文件)

设置Source Map以便可  
以在DevTools中显示  
正确的源代码行号

应用Babel, 它使用package.json  
中的Babel配置

这个 Webpack 配置文件与已经构建的其他项目中的文件没有区别。它设置了用于转换 JSX 文件的 Babel，并配置了打包后的文件的存放位置。

## 20.2 实现 Web 服务器

在这个项目中，不仅仅需要 HTML 主页，还需要一台简单的 Web 服务器，以便可以根据当前输入的内容来接收请求并响应(通过返回一个推荐列表)。它还将在服务器端渲染控件，并将相应的 HTML 发送给客户端。如前所述，这个示例使用 Express 作为 Web 服务器。index.js 文件定义了 Web 服务器，并包含如下三部分：

- 导入库和组件
- 定义用于接收请求的 REST API
- 在服务器端渲染控件

我们依次查看每个部分。首先是最直接的部分：导入库。代码清单 20.3 显示了服务器需要的组件和库(ch20/autocomplete/index.js)。

代码清单20.3 服务器需要的库和组件

```
const express = require('express'),
      mongodb = require('mongodb'),
      app = express(),
      bodyParser = require('body-parser'),
      validator = require('express-validator'),
      logger = require('morgan'),
      errorHandler = require('errorhandler'),
      compression = require('compression'),
      expHbs = require('express-handlebars'),
      url = 'mongodb://localhost:27017/autocomplete',
```

实例化 Express 应用

将MongoDB 连接字符串 设置为本地 数据库

使用逗号(多行)声明 定义和导入(类似于 每一行都有const)

```

ReactDOM = require('react-dom'),
ReactDOMServer = require('react-dom/server'),
React = require('react')

require('babel-register')({
  presets: ['react']
})
const Autocomplete = ,
React.createFactory(require('./src/autocomplete.jsx')),
port = 3000
...

```

定义babel-register  
预设来导入JSX文件

从JSX文件创建React组件工厂函数(将返回新的实例, 不需要使用createElement())

下一节继续介绍 index.js, 并讨论如何连接数据库和中间件。

## 20.2.1 定义 RESTful API

index.js 文件具有/rooms 的 GET 和 POST 路由。它们为前端应用提供 RESTful API 端点来提供数据。服务端数据来自 MongoDB 数据库, 可以使用 npm 脚本(npm run seed)查看数据(前提是在 package.json 中包含 seed 脚本, 并且拥有 rooms.json 文件)。但是在从数据库获取数据之前, 需要连接数据库并定义 Express 路由, 参见代码清单 20.4(ch20/autocomplete/index.js)。

代码清单20.4 RESTful API路由

```

mongodb.MongoClient.connect(url, function(err, db) {
  if (err) {
    console.error(err)
    process.exit(1)
  }
  app.use(compression())
  app.use(logger('dev'))
  app.use(errorHandler())
  app.use(bodyParser.urlencoded({extended: true}))
  app.use(bodyParser.json())
  app.use validator()
  app.use(express.static('public'))
  app.engine('handlebars', exphbs())
  app.set('view engine', 'handlebars')

  app.use(function(req, res, next){
    req.rooms = db.collection('rooms')
    return next()
  })

  app.get('/rooms', function(req, res, next) {
    req.rooms
      .find({}, {sort: {_id: -1}})
      .toArray(function(err, docs) {
        if (err) return next(err)
        return res.json(docs)
      })
  })

  app.post('/rooms', function(req, res, next) {
    req.checkBody('name', 'Invalid name in body')
      .notEmpty()

```

连接MongoDB

发生错误时终止当前进程

返回现有聊天室的列表

创建新的聊天室

验证body是否包含name字段, 并且name不为空



```
var errors = req.validationErrors()
if (errors) return next(errors)
req.rooms.insert(req.body, function (err, result) {
  if (err) return next(err)
  return res.json(result.ops[0])
})
})
```

调用数据库以保存新的消息

如果需要查看 Express API，可以查看附录 C 中的速查表。

20.2.2 在服务器端渲染 React

最后，index.js 包含/路由，在这个路由中通过使用 room 对象混合组件，在服务器上渲染 React，参见代码清单 20.5(ch20/autocomplete/index.js)。

代码清单20.5 服务器端React

```
app.get('/', function(req, res, next){
  var url = 'http://localhost:3000/rooms'
  req.rooms.find({}, {sort: {_id: -1}}).toArray(function(err, rooms){
    if (err) return next(err)
    res.render('index', {
      autocomplete: ReactDOMServer.renderToString(Autocomplete({
        options: rooms,
        url: url
      })),
      data: `<script type="text/javascript">
        window.__autocomplete_data = {
          rooms: ${JSON.stringify(rooms, null, 2)},
          url: "${url}"
        }
      </script>`
    })
  })
})
```

创建Autocomplete组件的React元素

传递rooms作为options属性的值

传递API的URL，用于获取和创建名称

将数据从服务器传递到浏览器，以确保通用React可以正常运行

使用stringify参数缩小输出

Autocomplete 组件有两个属性: options 和 url。options 包含聊天室的名称，url 是服务端 API 的 URL(在本例中是 http://localhost:3000/rooms)。Autocomplete 组件在浏览器端同样可以渲染。

20.3 添加浏览器脚本

浏览器脚本是用户如何使用自动完成小部件的示例，只能在浏览器上运行。文件内容很短，只需要创建一个包含 options 和 url 属性的元素，参见代码清单 20.6(ch20/autocomplete/src/app.jsx)。

代码清单20.6 客户端脚本

```
const React = require('react')
const ReactDOM = require('react-dom')
```

```

const Autocomplete = require('./autocomplete.jsx')
const {rooms, url} = window.__autocomplete_data

ReactDOM.render(<Autocomplete
  options={rooms}
  url={url}/>,
  document.getElementById('autocomplete')
)

```

获取来自全局变量的数据

使用已经存在的数据创建并渲染组件

其中,全局变量`__autocomplete_data`是通过/路由中的`local.data`(`local`是Express语言中模板数据的术语),使用`<script>`标签提供的,参见代码清单20.7。

#### 代码清单20.7 Express应用为浏览器提供React所需数据

```

res.render('index', {
  // ...
  data: `<script type="text/javascript">
    window.__autocomplete_data = {
      rooms: ${JSON.stringify(rooms, null, 2)},
      url: "${url}"
    }
  </script>`
})

```

使用script元素在Handlebars模板index.hbs中输出JavaScript

将数据从对象转换为字符串来输出

将`<script>` HTML 标签注入 `index.hbs` 模板(可自定义.hbs 文件扩展名)。接下来,我们将实现这个模板文件。

## 20.4 创建服务器端模板

在 `index.hbs` 文件中,可以看到正在输出的模板数据 `data` 和 `autocomplete`, 参见代码清单20.8。

#### 代码清单20.8 主机标记页面

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>Autocomplete with React.js</title>
    <meta name="description" content="React Quickly: Autocomplete" />
    <meta name="author" content="Azat Mardan" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <link type="text/css" rel="stylesheet" href="/css/bootstrap.css" />
  </head>

  <body>
    <div class="container-fluid">
      <div>{{{data}}}</div>
      <div class="row-fluid">
        <div class="span12">
          <div id="content">
            <div class="row-fluid" id="autocomplete">{{{autocomplete}}}</div>
          </div>
        </div>
      </div>
    </div>
  </body>
</html>

```

渲染包含名称列表和API的URL的<script>标签

使用通用 React 的校验和渲染静态HTML

```

<script type="text/javascript" src="/js/app.js"></script>
</body>
</html>

```

加载激活浏览器端React和使用  
\_\_autocomplete\_data\_\_的客户端  
脚本(请参阅上一节)

运行自动完成示例的工作已完成。很明显,这个示例的功能由 Autocomplete 组件提供。接下来,我们开始实现这个组件。

## 20.5 实现 Autocomplete 组件

Autocomplete 组件是自给自足的,意思是说它不仅仅是视图组件,还可以通过 REST API 获取和保存数据。它有两个属性: options 和 url。根据 TDD/BDD 原则,我们首先开始编写 Autocomplete 组件的测试用例。

### 20.5.1 Autocomplete 组件的测试

根据 TDD/BDD 原则,我们应该从测试开始。\_\_tests\_\_/autocomplete.test.js 文件包含房间名称列表,然后渲染组件并赋值给 autocomplete:

```

const rooms = [
  { "_id": "5622eb1f105807ceb6ad868b", "name": "node" },
  { "_id": "5622eb1f105807ceb6ad868c", "name": "react" },
  { "_id": "5622eb1f105807ceb6ad868d", "name": "backbone" },
  { "_id": "5622eb1f105807ceb6ad868e", "name": "angular" }
]

const TestUtils = require('react-addons-test-utils'),
  React = require('react'),
  ReactDOM = require('react-dom'),
  Autocomplete = require('../src/autocomplete.jsx'),
  fd = ReactDOM.findDOMNode

const autocomplete = TestUtils.renderIntoDocument(
  React.createElement(Autocomplete, {
    options: rooms,
    url: 'test'
  })
)

const optionName = TestUtils.findRenderedDOMComponentWithClass(autocomplete,
  'option-name')
...

```

硬编码房间  
名称列表

为方便起见,保存ID  
对象(少打字意味着更  
少的错误)

使用react-addons-test-utils  
的TestUtils渲染Autocomplete  
组件

通过类名option-name  
获取输入字段

通过类名 option-name 获取输入字段后,与房间名称进行匹配。

现在开始编写实际的测试,从小部件 autocomplete 中获取所有类名包含 option-list-item 的元素,然后比较元素数量是否等于 4(rooms 数组的长度,即房间数量):

```

describe('Autocomplete', () => {
  it('have four initial options', () => {
    var options = TestUtils.scrayRenderedDOMComponentsWithClass(
      autocomplete,
      'option-list-item'
    )
  })
})

```



```
expect(options.length).toBe(4)
})
```

接下来的测试更改输入字段中的内容，然后检查输入的字符以及组件根据所输入的字符匹配出的选项数量。当输入字符 `r` 后，应该仅有“`react`”选项与之匹配：

```
it('change options based on the input', () => {
  expect(fd(optionName).value).toBe('')
  fd(optionName).value = 'r'
  TestUtils.Simulate.change(fd(optionName))
  expect(fd(optionName).value).toBe('r')
  options = TestUtils.scrRenderedDOMComponentsWithClass(autocomplete,
    'option-list-item')
  expect(options.length).toBe(1)
  expect(fd(options[0]).textContent).toBe('#react')
})
```

最后改变输入字符为 `ember`，此时应该没有任何匹配，而是仅显示 `Add` 按钮：

```
it('offer to save option when there are no matches', () => {
  fd(optionName).value = 'ember'
  TestUtils.Simulate.change(fd(optionName))
  options = TestUtils.scrRenderedDOMComponentsWithClass(
    autocomplete,
    'option-list-item'
  )
  expect(options.length).toBe(0)
  var optionAdd = TestUtils.findRenderedDOMComponentWithClass(
    autocomplete,
    'option-add'
  )
  expect(fd(optionAdd).textContent).toBe('Add #ember')
})
})
```

## 20.5.2 Autocomplete 组件的代码

最后开始编写 `Autocomplete` 组件(ch20/autocomplete/src/autocomplete.jsx)，它包括输入框，匹配选项列表和用于在没有匹配项时添加新选项的 `Add` 按钮。这个组件有两个 `AJAX/XHR` 调用：获取选项列表和创建一个新的选项。它还有如下两个方法：

- `filter()`：<input> 字段每次发生变化都会调用。获取当前输入和选项列表，并将状态设置为仅包含与当前输入匹配的选项的新列表。
- `addOption()`：单点击 `Add` 按钮或按下回车键时调用。获取输入框中的值并将值发送到服务器。

从较高的抽象级别而言，`Autocomplete` 组件的代码如下：

```
const React = require('react'),
      ReactDOM = require('react-dom'),
      request = require('axios')

class Autocomplete extends React.Component {
  constructor(props) {
```

```

    ...
  }
  componentDidMount() {
    ...
  }
  filter(event) {
    ...
  }
  addOption(event) {
    ...
  }
  render() {
    return (
      <div ...>
        <input ... onChange={this.filter}>
        </input>
        {this.state.filteredOptions.map(function(option,
          index, list) {
            ...
          })}
        <a ...onClick={this.addOption}>
          Add #{this.state.currentOption}
        </a>
        ...
      </div>
    )
  }
}

module.exports = Autocomplete

```

从服务器获取选项列表

过滤列表，只留下与输入匹配的选项

通过XHR请求添加新的选项

通过浏览器事件捕获输入值

输出匹配的列表

单击按钮时调用添加方法

现在从这个文件的开头开始讲解。首先通过 CommonJS/Node.js 模块规范导入相关库，由于 Webpack 提供对 CommonJS 规范的支持，使得浏览器可以运行该文件。fd 别名是为了方便使用：

```

const React = require('react'),
      ReactDOM = require('react-dom'),
      request = require('axios')

const fd = ReactDOM.findDOMNode

```

constructor 构造函数设置状态和函数绑定。从组件属性设置 options。filteredOptions 的初始值和 options 相同，而且当前输入字段也为空。当用户输入字符时，filteredOptions 将变得越来越小，以匹配输入的字母。

在 componentDidMount() 中，使用 axios(request 变量)库执行 GET 请求。它类似于 jQuery 的 \$.get()，但 axios 是基于 promise 的 HTTP 库：

```

class Autocomplete extends React.Component {
  constructor(props) {
    super(props)
    this.state = {options: this.props.options,
      filteredOptions: this.props.options,
      currentOption: ''
    }
    this.filter = this.filter.bind(this)
  }
}

```

```

    this.addOption = this.addOption.bind(this)
  }
  componentDidMount() {
    if (this.props.url == 'test') return true
    request({url: this.props.url})
      .then(response=>response.data)
      .then(body => {
        if(!body){
          return console.error('Failed to load')
        }
        this.setState({options: body})
      })
      .catch(console.error)
  }
  ...

```

测试时阻止发送请求

给options设置结果

每当输入字段发生变化时，调用 `filter()` 方法，目的是仅留下匹配用户输入的选项：

```

...
filter(event) {
  this.setState({
    currentOption: event.target.value,
    filteredOptions:
      (this.state.options.filter((option, index, list) => {
        return (event.target.value === option.name.substr(0,
          event.target.value.length))
      })))
  })
}

```

对数组使用 `filter()`

截断超出字符

当没有匹配时，`addOption()` 方法处理新选项的添加事件：

```

addOption(event) {
  let currentOption = this.state.currentOption
  request
    .post(this.props.url, {name: currentOption})
    .then(response => response.data)
    .then((body) => {
      if(!body){
        return console.error('Failed to save')
      }
      this.setState({
        options: [body].concat(this.state.options)
      },
        () => {
          this.filter({target: {value: currentOption}})
        }
      )
    })
    .catch(error=>{return console.error('Failed to save')})
}

```

使用axios发送POST请求

使用 `Array.concat()` 而不是 `Array.push()` 创建一个新的数组，因为直接改变状态是不好的做法

在 `setState()` 的回调中调用 `filter()` 方法，以确保 `filter()` 运行时的状态值是最新的

最后，`render()` 方法中有一个带有 `onChange` 事件监听器 `this.filter` 的受控组件 `<input>`：

```

...
render() {
  return (
    <div className="form-group">
      <input type="text"

```



```

    onKeyUp={(event) => (event.keyCode===13) ? this.addOption() : ''}
    className="form-control option-name"
    onChange={this.filter}
    value={this.currentOption}
    placeholder="React.js">
  </input>

```

onKeyUp 可以在 {} 中写成一种方法，不一定作为匿名内联函数。

已过滤的选项列表由 state.filtersOptions 提供，这个列表在 filter() 方法中更新。迭代 filtersOptions，将 \_id 作为 key，将 option.name 作为链接锚点的值：

```

    {this.state.filteredOptions.map(function(option, index, list){
      return <div key={option._id}>
        <a className="btn btn-default option-list-item"
          href={'/'+option.name} target="_blank">
          #{option.name}
        </a>
      </div>
    })}
    ...

```

使用 map() 方法显示过滤选项的列表

显示带有井号(#)的选项名称，就像 Slack 应用一样

使用 URL 作为每个选项的锚点标签的值

最后一个元素是 Add 按钮，这个按钮仅在没有任何匹配选项时才显示：

```

    ...
    {(()=>{
      if (this.state.filteredOptions.length == 0 &&
        this.state.currentOption!='')
        return <a className="btn btn-info option-add"
          onClick={this.addOption}>
            Add #{this.state.currentOption}
          </a>
    })()}
  </div>
)
}

```

提示添加当前输入的值作为选项

有匹配时隐藏按钮

使用 addOption 作为 onClick 事件处理程序

项目使用 CommonJS 语法，因此使用如下方式声明 Autocomplete 组件并将其导出：

```
module.exports=Autocomplete
```

## 20.6 整合

如果遵循以上步骤，应该可以使用如下命令安装依赖(如果还没有安装的话)：

```
$ npm install
```

然后启动应用(启动前，需要使用 \$ mongod 启动 MongoDB)：

```
$ npm start
```

运行如下命令后，测试将通过：

```
$ npm test
```

还要运行 `npm run build`, 但是不能监听文件变化(文件改变时需要重新运行命令)。`npm start` 会为你运行 `npm run build`。

作为可选项, 可以使用 `$ npm run seed` 将 `ch20/autocomplete/rooms.json` 中的数据填充到数据库中:

```
[{"name": "react"},
{"name": "node"},
{"name": "angular"},
{"name": "backbone"}]
```

这就是 Autocomplete 组件。现在, 通过使用 `npm run build` 构建项目并在浏览器中打开 `http://localhost:3000`(前提是 MongoDB 已在单独的终端运行)。虽然 `127.0.0.1` 是一个别名, 但还是必须使用与浏览器相同的域 `localhost` 来避免 CORS/Access-Control-Allow-Origin 问题, 因为 JavaScript 会调用 `localhost` 服务器。

这时应该可以在页面上看到具有名称的组件(如果已将数据导入数据库的话)。当在输入字段中输入字符时, 将根据输入中的匹配来过滤选项。当没有匹配时, 单击 Add 按钮将房间添加到数据库中, 并且新添加的选项会立即出现在列表中。

### Mongo 和 MongoUI

如果需要直接操作 MongoDB 中的数据, 可以通过终端中的 `mongo` 命令使用 `mongo shell`(又名 REPL)。它会自动连接到运行在端口 27017 上的本地实例(必须有一个处在运行状态, 如果没有, 可以使用 `mongod`)。在 `mongo shell` 中, 可以执行各种操作, 如创建新文档、查询集合、删除数据库等。优点是可以在任何地方使用 `mongo shell`, 即使在没有 GUI 的远程服务器上也可以。

但是在使用 `mongo shell` 时会涉及很多输入, 而且打字速度慢、容易出错。因此, 我创建了一个更好的工具, 名叫 `MongoUI`(<https://github.com/azat-co/mongoui>), 借助这个工具, 可以通过单击触控板进行查询, 编辑、添加和删除文档, 以及在浏览器中执行其他操作, 而不是输入大量的 JSON 文本(MongoDB 所存储数据的格式为 BSON, 是 JSON 的一种扩展)。

MongoUI 允许通过一个友好的 Web 界面来使用 MongoDB。图 20.6 显示了 `autocomplete` 数据库中的 `rooms` 集合数据。

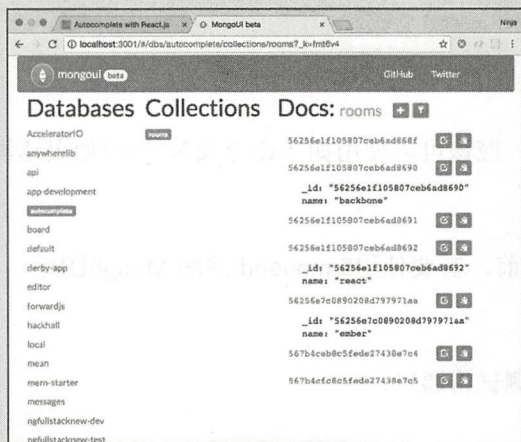


图 20.6 MongoDB 的 Web 页面



使用 `npm i -g mongoui` 安装 MongoUI 并运行命令 `mongoui` 来启动,然后在浏览器中打开 `http://localhost:3001`。另外, MongoUI 是使用 React、Express 和 Webpack 开发的。

Autocomplete 示例的最终结果如图 20.7 所示。可以打开 Network 选项卡,然后单击 `localhost`,以确保服务器端渲染可以正常工作(即查看 `names` 对应的数据和 HTML)。



图 20.7 通过单击 Network 和 localhost 来检查 localhost 响应,以确保服务器端渲染正常工作

如果由于某些原因导致项目无法工作,很有可能是因为新版本或者输入错误的代码。可以访问链接 [www.manning.com/books/react-quickly](http://www.manning.com/books/react-quickly) 或 <https://github.com/azat-co/react-quickly/tree/master/ch20> 以查看最新可运行的代码。

## 20.7 测验

为了有更多收获,不妨尝试下面的操作:

- 添加用于 Remove 按钮的测试,即在每个选项名称旁添加 X 图标。
- 将 Remove 按钮添加到每个选项名称旁的 X 图标上。实现一个 AJAX/XHR 调用,并添加一个 REST 端点来处理删除操作。
- 增强匹配算法,使其可以在名称中间找到匹配项。例如,输入 `ac` 后应该显示 `react` 和 `backbone`,因为它们都包含字母 `ac`。



- 添加 Redux 数据存储。
- 使用 GraphQL 替换 REST API。

将代码提交到本书的 GitHub 仓库(<https://github.com/azat-co/react-quickly/>)中 ch20 目录下的新文件夹中。

## 20.8 小结

- 在 Handlebars 中使用花括号输出未转义的 HTML，而在 React 中需要使用 `__html` 设置 `dangerouslySetInnerHTML` 属性。
- `findRenderedDOMComponentWithClass()` 通过 CSS 类名找到单个组件，而 `scryRenderedDOMComponentsWithClass()` 通过 CSS 类名查找多个组件(参见第 16 章)。
- `babel-register` 可以让你导入和使用 JSX 文件：`require('babel-register')({presets: ['react']})`。
- `MongoUI` 是一个开源的、使用 React 开发的用于开发和管理 MongoDB 数据库的 Web 界面。可以使用 `npm i -g mongoui` 来安装，并使用 `mongoui` 来运行。

# 附录

# A

## 安装本书相关应用

本附录将介绍如下应用的安装步骤(截至 2017 年 5 月均有效):

- React v15
- Node.js v6 和 npm v3
- Express v4
- Twitter Bootstrap v3
- Browserify
- MongoDB
- Babel

### 安装 React

可通过如下任意方法下载:

- 直接通过 CDN 资源链接下载, 如 <https://cdnjs.cloudflare.com/ajax/libs/react/15.5.4/react.js> 或 <https://cdnjs.cloudflare.com/ajax/libs/react/15.5.4/react-dom.js>(全部链接列表: <https://cdnjs.com/libraries/react>)。
- 通过 React 官网下载, 如 <http://facebook.github.io/react/downloads.html> 或 <https://github.com/facebook/react>。
- 使用 `npm install react@15 react-dom@15` 来安装(下一小节介绍 npm)。现在不需要关心在服务器上渲染 React。react.js 在目录 `node_modules/react/dist` 中。
- 使用 `bower install --save react` 来安装。
- 使用 Webpack/Grunt/Browserify/Gulp 从 npm 模块中自行构建。

## 安装 Node.js

如果不确定是否安装了 Node.js 和 npm，或者不清楚已安装的版本，可以在终端 (Terminal/iTerm/bash/zsh 等) 中使用如下命令查看版本信息：

```
$ node -v
$ npm -v
```

一般情况下，安装 Node.js 时会自动安装 npm。安装 Node.js 和 npm 的最简单方法是通过官网(<https://nodejs.org/en/download>)下载特定操作系统(Windows 或 Mac OS 等)的版本。

对于已经安装 Ruby(通常情况下 Mac OS 已自带 Ruby)的 Mac OS 用户，强烈推荐使用 Homebrew。这也是我正在使用的，因为还可以用它安装其他开发工具，如数据库和服务器。要在 Mac OS 上获取 brew，请在终端中运行如下 Ruby 代码(我保证这是最后一次在本书中使用 Ruby)：

```
$ ruby -e "$(curl -fsSL
➤ https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

现在应该已经安装了 brew。更新它的注册表，并安装 Node.js 和 npm。就像前面提到的那样，npm 会随同 Node.js 一起安装，不需要额外的命令：

```
$ brew update
$ brew install node
```

另外推荐一个非常好用的工具 nvm(<https://github.com/creationix/nvm>)，它可以用来自由切换 Node.js 的版本：

```
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.32.1/install.sh |
➤ bash
$ nvm install node
```

现在应该可以查看 Node 和 npm 的版本信息。如果想要升级 npm，可以使用如下 npm 命令：

```
$ npm i -g npm@latest
```

可以使用 nvm 或其他(如 nave 或 n)类似应用来升级 Node。例如，使用如下 nvm 命令重新安装 Node 到最新版本：

```
$ nvm install node --reinstall-packages-from=node
```

如果使用 npm 安装模块/包时，提示权限错误，可以使用如下命令更改权限(确保在运行之前了解此命令的功能)：

```
$ sudo chown -R $USER /usr/local/{share/man,bin,lib/node,include/node}
```



## 安装 Express

Express 就像 React 一样，是一个本地依赖，这意味着每个项目都必须安装它。安装 Express 的唯一方法是使用 npm:

```
npm i express@4.13.3 -S
```

-S 的意思是将 Express 信息写入 package.json。

这里绝不是深入了解 Express，但会让你开始使用应用最广泛的 Node.js Web 框架。首先，使用如下 npm 命令安装 Express:

```
$ npm install express@4.13.3
```

通常来讲，需要服务器文件 index.js、app.js 或 server.js，稍后 node 命令将使用该文件启动程序。该文件包含如下部分:

- 导入
- 配置
- 中间件
- 路由
- 错误捕获
- 启动

导入部分很简单。其中，需要导入依赖并实例化对象。例如，使用如下代码导入 Express 框架并创建一个实例:

```
var express = require('express')
var app = express()
```

在配置部分使用 app.set() 进行配置设置，app.set() 函数接收两个参数：第一个参数是字符串；第二个参数是值。例如，使用 view engine 设置 Jade 为模板引擎:

```
app.set('view engine', 'jade')
```

下一部分是中间件，中间件与插件类似，例如使用 static 中间件设置服务器静态资源路径:

```
app.use(express.static(path.join(__dirname, 'public')))
```

最重要的是路由部分，使用 app.NAME() 模式定义路由。例如，以下是来自 ch20/autocomplete 的 GET /rooms 路由的代码:

```
app.get('/rooms', function(req, res, next) {
  req.rooms.find({}, {sort: {_id: -1}}).toArray(function(err, docs){
    if (err) return next(err)
    return res.json(docs)
  })
})
```

错误捕获类似于中间件:

```
var errorHandler = require('errorhandler')
app.use(errorHandler)
```

最后运行 `listen()` 来启动服务:

```
http.createServer(app).listen(portNumber, callback)
```

当然, Express 涉及的内容比上述简短介绍要多得多。否则, 我也不会为这个框架专门写一本 350 页的书(*Pro Express.js*; Apress, 2014, <http://proexpressjs.com>)。如果想听听不同作者的观点, 可以参考 Evan M. Hahn 的著作 *Express in Action*(Manning, 2016, [www.manning.com/books/express-in-action](http://www.manning.com/books/express-in-action))。该框架功能强大且灵活, 可以在不需要太多技巧的情况下进行配置。

如果构建 Express 应用不是你的核心能力, 或者你知道如何做, 但需要复习, 可以查看附录 C 中的 Express 速查表或通过 <http://reactquickly.co/resources/> 查看图形版本。

## 安装 Bootstrap

可以打开官网(<http://getbootstrap.com>)来查看 Twitter Bootstrap, 本书使用 v3.3.5 版本, 可以使用如下任意方式进行安装:

- 下载不带文档的压缩的 JavaScript 和样式文件的压缩包, 无须修改即可使用:  
<https://github.com/twbs/bootstrap/releases/download/v3.3.5/bootstrap-3.3.5-dist.zip>。
- 下载源代码的 Less 版(<https://github.com/twbs/bootstrap/archive/v3.3.5.zip>)或 Sass 版(<https://github.com/twbs/bootstrap-sass/archive/v3.3.5.tar.gz>)。若需要更改文件, 这两种方式是比较理想的。
- 直接使用 CDN 链接。由于缓存, 可以获得更好的性能, 但是这种方式需要网络连接。
- 使用 Bower 安装 Bootstrap。
- 使用 npm 安装 Bootstrap。
- 使用 Composer 安装 Bootstrap。
- 通过只选择需要的组件来创建自己的 Bootstrap 版本: <http://getbootstrap.com/customize>
- 不需要做太多工作就可以使用 Bootstrap 主题来更换外观。例如, Bootswatch 在 <https://bootswatch.com> 上提供 Bootstrap 主题。

要使用 CDN 链接, 请在 HTML 文件中包含如下标签:

```
<!-- Latest compiled and minified CSS -->
<link rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/bootstrap.min.css">

<!-- Optional theme -->
<link rel="stylesheet"
      href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/css/
      bootstrap-theme.min.css">

<!-- Latest compiled and minified JavaScript -->
<script src=
      "https://maxcdn.bootstrapcdn.com/bootstrap/3.3.5/js/bootstrap.min.js">
</script>
```

在项目目录中，为 Bower、npm 或 Composer 执行如下命令安装 Bootstrap：

```
$ bower install bootstrap
$ npm install bootstrap
$ composer require twbs/bootstrap
```

更多信息请访问 <http://getbootstrap.com/getting-started>。

## 安装 Browserify

Browserify 可以将 npm 模块打包到前端软件包中，以便在浏览器中使用。基本上，可以将任何 npm 模块(通常只用于 Node)转换为前端模块。

注意：如果使用 Webpack，就不需要 Browserify。

首先使用 npm 安装 Browserify：

```
$ npm install -g browserify
```

我们使用 `ch16/jest` 作为示例，进入该文件夹并新建文件 `scripts.js`，在该文件中引入 `generate-password.js`。`scripts.js` 中的内容如下所示：

```
var generatePassword = require('generate-password')
console.log(generatePassword())
console.log(generatePassword())
```

保存 `scripts.js`，并在终端或命令提示符下运行如下命令：

```
$ browserify script.js -o bundle.js
```

查看 `bundle.js`，并在 `index.html` 中引入如下内容：

```
<script src="bundle.js"></script>
```

在浏览器中打开 `index.html` 文件，并查看控制台，控制台会显示两个随机密码。源代码在文件 `ch16/jest` 中。

## 安装 MongoDB

安装 MongoDB 的最简单方法是访问官网([www.mongodb.org/downloads#production](http://www.mongodb.org/downloads#production))，并选择适合系统版本的软件包。

在 Mac OS 系统中，可以使用 `brew` 运行如下命令进行安装：

```
$ brew update
$ brew install mongodb
```

不要用 npm 全局安装 `mongodb`，npm 的 `mongodb` 是驱动程序，不是数据库，所以它属于本地 `node_modules` 文件夹中的其他依赖。



本书使用 3.0.6 版本，因此，使用更新(或更早)的版本时请自行承担风险，本书示例并没有在其他版本上进行过测试。

通常，需要创建一个具有适当权限的/data/db 文件夹。也可以使用如下 `mongod --dbpath` 命令自定义文件夹，例如：

```
$ mongod -dbpath ./data
```

数据库运行后，在 shell 中执行 `mongo`：

```
$ mongo
> 1+1
> use autocomplete
> db.rooms.find()
```

以下是对一些常用 shell 命令的解释：

- `> show dbs`：显示服务器上的数据库列表。
- `> use DB_NAME`：选择 `DB_NAME` 数据库。
- `> show collections`：显示所选数据库中的集合。
- `> db.COLLECTION_NAME.find()`：对名为 `COLLECTION_NAME` 的集合执行查询，以查找任意项。
- `> db.COLLECTION_NAME.find({"_id": ObjectId("549d9a3081d0f07866fdaac6")})`：对名为 `COLLECTION_NAME` 的集合执行查询，查找 `id` 为 `549d9a3081d0f07866fdaac6` 的项。
- `> db.COLLECTION_NAME.find({"email": /gmail/})`：对名为 `COLLECTION_NAME` 的集合执行查询，查找 `email` 属性匹配 `/gmail/` 的项。
- `> db.COLLECTION_NAME.update(QUERY_OBJECT, SET_OBJECT)`：对名为 `COLLECTION_NAME` 的集合执行查询，查找匹配 `QUERY_OBJECT` 的项并更新为 `SET_OBJECT`。
- `> db.COLLECTION_NAME.remove(QUERY_OBJECT)`：对 `COLLECTION_NAME` 集合中符合 `QUERY_OBJECT` 条件的项执行删除操作。
- `> db.COLLECTION_NAME.insert(OBJECT)`：将 `OBJECT` 添加到名为 `COLLECTION_NAME` 的集合中。

在附录 D 中查看 MongoDB 速查表，或在 <http://reactquickly.co/resources> 上查看 MongoDB 的图形版本。除了最常用的 MongoDB 命令外，还包括 Mongoose(Node.js ODM)方法。

## 使用 Babel 编译 JSX 和 ES6

Babel 主要用于 ES6+/ES2015+，但也可以将 JSX 转换为 JavaScript。通过在 React 中使用 Babel，可以获得额外的 ES6 功能来简化开发。

ES6 已经定稿，但它的功能以及 ECMAScript 未来版本的功能可能并不被所有浏览器完全支持。要使用未被支持的新功能(如 ES Next(<https://github.com/esnext/esnext>))或在旧版浏览器(IE9)中使用 ES6，需要使用 Babel(<https://babeljs.io>)进行编译。Babel 可以作为独立

工具运行或与构建系统一起使用。

要使用 Babel 作为独立的 CLI 工具, 首先创建一个新的文件夹。如果已经安装了 Node.js 和 npm, 请运行如下命令来创建 package.json:

```
$ npm init
```

打开 package.json 文件, 并在 JSON 文件中添加 babel 代码行, 可以任意顺序添加, 因为 babel 是顶层属性。添加 babel 设置是为了告诉 Babel 使用 React 和 JSX 来转换源文件。babel 设置被称为预设, 没有它, Babel CLI 不会做任何事情:

```
"babel": {  
  "presets": ["react"]  
},
```

使用 npm 安装 Babel CLI v6.9.0 和 React preset v6.5.0, 在终端、命令提示符或 shell 中, 执行以下命令:

```
$ npm i babel-cli@6.9.0 --save-dev  
$ npm i babel-preset-react@6.5.0 --save-dev
```

可以使用如下命令查看 Babel 版本信息:

```
$ babel --version
```

有针对 Grunt、Gulp 和 Webpack 的 Babel 插件(<http://babeljs.io/docs/setup>)。这是一个 Gulp 示例, 使用以下命令安装插件:

```
$ npm install --save-dev gulp-babel
```

在 gulpfile.js 中, 定义将 src/app.js 编译到构建文件夹中的 build 任务:

```
var gulp = require('gulp'),  
    babel = require('gulp-babel')  
gulp.task('build', function () {  
  return gulp.src('src/app.js')  
    .pipe(babel())  
    .pipe(gulp.dest('build'))  
})
```

Webpack 和 Babel 的更多信息请查看第 12 章。

## Node.js 和 ES6

可以使用构建工具编译 Node.js 文件或使用独立的 Babel 模块 babel-core。安装方式如下:

```
$ npm install --save-dev babel-core@6
```

然后在 Node.js 中调用此函数:

```
require('babel-core').transform(es5Code, options)
```

## 独立的浏览器版本 Babel

Babel v5.x 具有独立的浏览器文件，可用于在浏览器中进行转换(仅限于开发)。这个独立的文件在 6.x 版本中被删除，但有人创建了一个 `babel-standalone` 模块来填补空白(<https://github.com/Daniel15/babel-standalone>)。可以直接使用这个模块或旧版本的文件，例如，从 Cloudflare CDN 获取：

- 未压缩版：<http://mng.bz/klbg>
- 压缩版：<http://mng.bz/sM59>

也可以使用构建工具(如 Gulp 或 Webpack)构建自己独立的浏览器文件。通过这种方式，可以选择自己需要的内容，比如 React transformer 插件和 ES2015 预设。



## 附录 B

## React 速查表

开发项目时，在互联网上搜索 React 文档和 API，或者回到本书的某一章，查找一个方法，这些方式的效率并不高。如果想节省时间，避免网络上的其他东西使你分心，建议使用本附录的 React 速查表作为快速参考。

## 打印 PDF 版本

除了这里介绍的文本版本之外，我还创建了一个免费的设计精美的 PDF 版本。可以通过访问 <http://reactquickly.co/resources> 下载 PDF 版本，如图 B.1 所示。

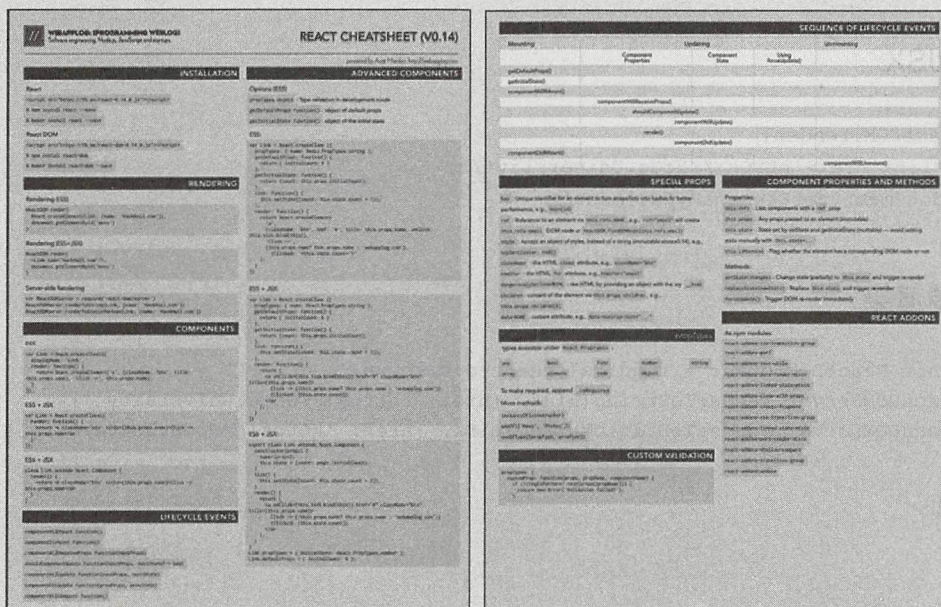


图 B.1 PDF 版本的 React 速查表



## 安装

### React

- `<script src="https://unpkg.com/react@15/dist/react.js"></script>`
- `$ npm install react --save`
- `$ bower install react --save`

### React DOM

- `<script src="https://unpkg.com/react-dom@15/dist/react-dom.js"></script>`
- `$ npm install react-dom`
- `$ bower install react-dom --save`

## 渲染

### ES5

```
ReactDOM.render(  
  React.createElement(  
    Link,  
    {href: 'https://Node.University'}  
  ),  
  document.getElementById('menu')  
)
```

### ES5+JSX

```
ReactDOM.render(  
  <Link href='https://Node.University' />,  
  document.getElementById('menu')  
)
```

### 服务端渲染

```
const ReactDOMServer = require('react-dom/server')  
ReactDOMServer.renderToString(Link, {href: 'https://Node.University'})  
ReactDOMServer.renderToStaticMarkup(Link, {href: 'https://Node.University'})
```

## 组件

### ES5

```
var Link = React.createClass({
```



```
    displayName: 'Link',
    render: function() {
      return React.createElement('a',
        {className: 'btn', href: this.props.href}, 'Click ->', this.props.href)
    }
  })
```

## ES5+JSX

```
var Link = React.createClass({
  render: function() {
    return <a className='btn' href={this.props.href}>Click ->
      this.props.href</a>
  }
})
```

## ES6+JSX

```
class Link extends React.Component {
  render() {
    return <a className='btn' href={this.props.href}>Click ->
      this.props.href</a>
  }
}
```

# 高级组件

## 选项(ES5)

- 开发模式下的属性类型检测: propTypes 对象
- 默认属性的对象: getDefaultProps function()
- 初始状态的对象: getInitialState function()

## ES5

```
var Link = React.createClass ({
  propTypes: { href: React.PropTypes.string },
  getDefaultProps: function() {
    return { initialCount: 0 }
  },
  getInitialState: function() {
    return {count: this.props.initialCount}
  },
  tick: function() {
    this.setState({count: this.state.count + 1})
  }
})
```





```

render: function() {
  return React.createElement(
    'a',
    {className: 'btn', href: '#', href: this.props.href,
      onClick: this.tick.bind(this)},
    'Click ->',
    (this.props.href ? this.props.href : 'https://webapplog.com'),
    ' (Clicked: ' + this.state.count+')'
  )
}
})

```

## ES5+JSX

```

var Link = React.createClass ({
  propTypes: { href: React.PropTypes.string },
  getDefaultProps: function() {
    return { initialCount: 0 }
  },
  getInitialState: function() {
    return {count: this.props.initialCount};
  },
  tick: function() {
    this.setState({count: this.state.count + 1})
  },
  render: function() {
    return (
      <a onClick={this.tick.bind(this)} href="#" className="btn"
        href={this.props.href}>
        Click -> {(this.props.href ? this.props.href :
          'https://webapplog.com')}
        (Clicked: {this.state.count})
      </a>
    )
  }
})

```

## ES6+JSX

```

export class Link extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  tick() {
    this.setState({count: this.state.count + 1});
  }
  render() {
    return (
      <a onClick={this.tick.bind(this)} href="#" className="btn"

```



```

      href={this.props.href}>
        Click -> {(this.props.href ? this.props.href :
          'https://webapplog.com')}
        (Clicked: {this.state.count})
      </a>
    )
  }
}
Link.propTypes = { initialCount: React.PropTypes.number }
Link.defaultProps = { initialCount: 0 }

```

## 生命周期事件

- componentWillMount function()
- componentDidMount function()
- componentWillReceiveProps function(nextProps)
- shouldComponentUpdate function(nextProps, nextState)→bool
- componentWillUpdate function(nextProps, nextState)
- componentDidUpdate function(prevProps, prevState)
- componentWillUnmount function()

生命周期事件的顺序(受 <http://react.tips> 启发)，如表 B.1 所示。

表 B.1 生命周期事件的顺序

挂载	更新组件属性	更新组件状态	使用forceUpdate()	卸载
getDefaultProps()				
getInitialState()				
componentWillMount()				
	componentWillReceiveProps()			
	shouldComponentUpdate()	shouldComponentUpdate()		
	componentWillUpdate()	componentWillUpdate()	componentWillUpdate()	
render()	render()	render()	render()	
	componentDidUpdate()	componentDidUpdate()	componentDidUpdate()	
componentDidMount()				componentWillUnmount()

## 特殊属性

- key: 数组或列表元素的唯一标识符，用来获得更好的性能。例如：key={id}。
- ref: 通过 this.refs.NAME 引用元素。例如：ref="email"会创建 this.refs.email DOM 节点，ReactDOM.findDOMNode(this.refs.email)可以获取真实 DOM 节点。
- style: 接收采用驼峰命名规则的 CSS 样式的对象而不是字符串(从 v0.14 版本开始)。例如：style={{color:red}}。
- className: HTML 元素的 class 属性。例如：className="btn"。



- `htmlFor`: HTML 元素的属性。例如: `htmlFor="email"`。
- `dangerouslySetInnerHTML`: 通过使用键 `__html` 提供对象, 将内部 HTML 设置为原生 HTML。
- `children`: 通过 `this.props.children` 设置元素内容。例如: `this.props.children[0]`。
- `data-NAME`: 自定义属性。例如: `data-tooltip-text="..."`。

## 属性类型

`React.PropTypes` 可用类型如下:

- `any`
- `array`
- `bool`
- `element`
- `func`
- `node`
- `number`
- `object`
- `string`

如果需要设置某个属性必须存在, 则可以添加后缀 `isRequired`。

更多方法如下:

- `instanceOf(constructor)`
- `oneOf(['News', 'Photos'])`
- `oneOfType([propTypes, propTypes])`

## 自定义验证器

```
propTypes: {
  customProp: function(props, propName, componentName) {
    if (!/regExpPattern/.test(props[propName])) {
      return new Error('Validation failed!');
    }
  }
}
```

## 组件属性和方法

### 属性

- `this.refs`: 列出具有 `ref` 属性的组件。
- `this.props`: 列出传递给元素的属性(不可变)。





- `this.state`: 列出由 `setState` 和 `getInitialState` 设置的组件状态。避免使用 `this.state=...` 直接赋值。
- `this.isMounted`: 判断组件是否还挂载在 DOM 节点上。

## 方法

- `setState(changes)`: 改变部分状态并触发渲染。
- `replaceState(newState)`: 替换全部状态并触发渲染。
- `forceUpdate()`: 强制触发渲染。

## React 插件

以下这些插件作为独立的 npm 模块:

- `react-addons-css-transition-group`(<http://facebook.github.io/react/docs/animation.html>)
- `react-addons-perf`(<http://facebook.github.io/react/docs/perf.html>)
- `react-addons-test-utils`(<http://facebook.github.io/react/docs/test-utils.html>)
- `react-addons-pure-render-mixin`(<http://facebook.github.io/react/docs/pure-render-mixin.html>)
- `react-addons-linked-state-mixin`(<http://facebook.github.io/react/docs/two-way-binding-helpers.html>)
- `react-addons-clone-with-props`
- `react-addons-create-fragment`
- `react-addons-css-transition-group`
- `react-addons-linked-state-mixin`
- `react-addons-pure-render-mixin`
- `react-addons-shallow-compare`
- `react-addons-transition-group`
- `react-addons-update`(<http://facebook.github.io/react/docs/update.html>)

## React 组件

- React 组件列表: <https://github.com/brillout/awesome-react-components> 和 <http://devarchy.com/react-components>。
- Material-UI: 符合 Material Design 规范的 React 组件(<http://material-ui.com>)。
- React Toolbox: Google Material Design 规范的 React 组件的实现(<http://react-toolbox.com>)。
- JS.Coach: 开源(大部分是 React 相关的)JS 包推荐(<https://js.coach>)。
- React Rocks: React 组件目录(<https://react.rocks>)。
- Khan Academy: 可复用 React 组件的集合(<https://khan.github.io/react-components>)。
- ReactJSX.com: React 组件仓库(<http://reactjsx.com>)



## 附录

## Express 速查表

开发项目时，在互联网上搜索 React 文档和 API，或者回到本书的某一章，查找一个方法，这些方式的效率并不高。如果想节省时间，避免网络上的其他东西使你分心，建议使用本附录的 Express 速查表作为快速参考。

## 打印 PDF 版本

除了这里介绍的文本版本之外，我还创建了一个免费的设计精美的 PDF 版本。可以通过访问 <http://reactquickly.co/resources> 下载 PDF 版本，如图 C.1 所示。

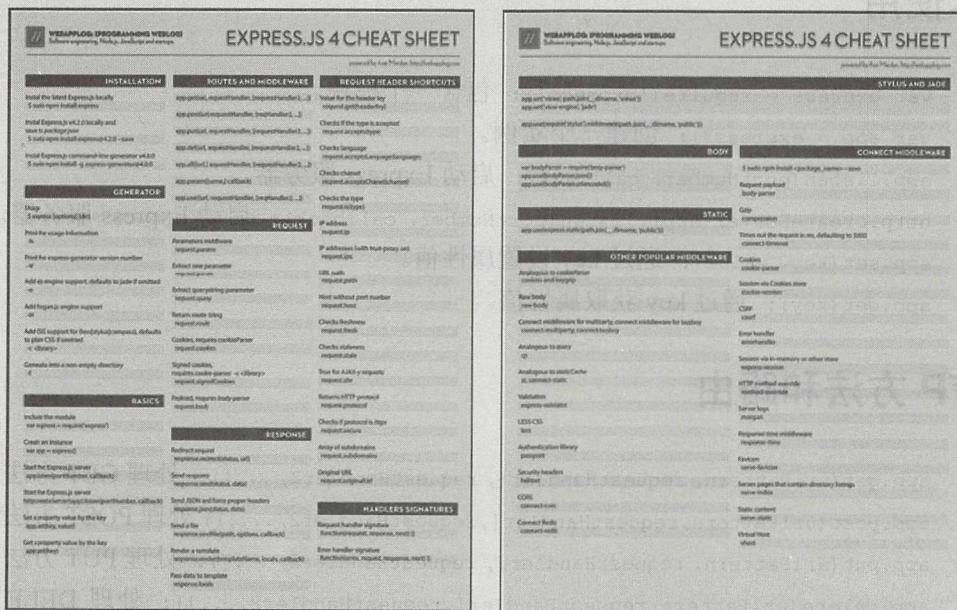


图 C.1 PDF 版本的 Express 速查表



## 安装 Express

- `$ sudo npm install express`: 本地安装最新版 Express。
- `$ sudo npm install express@4.2.0 --save`: 本地安装 4.2.0 版本的 Express 并添加至 `package.json` 中。
- `$ sudo npm install -g express-generator@4.0.0`: 全局安装 Express 应用生成器。

## 生成器

### 用法

```
$ express [options] [dir]
```

### 选项

- `-h`: 输出帮助信息。
- `-V`: 输出 `express-generator` 版本号。
- `-e`: 添加 EJS 模板引擎支持, 默认是 Jade。
- `-H`: 添加 hogan.js 模板引擎支持。
- `-c <library>`: 添加对 `<library>` (`less` | `stylus` | `compass`) 的 CSS 支持, 默认仅支持纯 CSS
- `-f`: 强制在非空目录中执行。

## 基本使用

- `var express = require('express')`: 包含一个模块。
- `var app = express()`: 创建一个实例。
- `app.listen(portNumber, callback)`: 启动 Express 服务器。
- `http.createServer(app).listen(portNumber, callback)`: 启动 Express 服务器。
- `app.set(key, value)`: 通过 `key` 设置属性值。
- `app.get(key)`: 通过 `key` 获取属性值。

## HTTP 方法和路由

- `app.get(urlPattern, requestHandler[, requestHandler2, ...])`: 处理 GET 方法请求。
- `app.post(urlPattern, requestHandler[, requestHandler2, ...])`: 处理 POST 方法请求。
- `app.put(urlPattern, requestHandler[, requestHandler2, ...])`: 处理 PUT 方法请求。
- `app.delete(urlPattern, requestHandler[, requestHandler2, ...])`: 处理 DELETE 方法请求。





- `app.all(urlPattern, requestHandler[, requestHandler2, ...])`: 处理所有方法请求。
- `app.param([name,]callback)`: 处理 URL 参数。
- `app.use([urlPattern,]requestHandler[,requestHandler2,...])`: 应用中间件。

## 请求

- `request.params`: 请求参数。
- `request.param`: 提取一个参数。
- `request.query`: 提取查询字符串参数。
- `request.route`: 返回一个路由字符串。
- `request.cookies`: 访问 cookies; 需要 `cookie-parser` 中间件。
- `request.signedCookies`: 访问已签名的 cookie; 需要 `cookie-parser` 中间件。
- `request.body`: 读取有效负载信息, 需要 `body-parser` 中间件。

## 访问请求头信息快捷方式

- `request.get(headerKey)`: 读取 `headerKey` 对应的值。
- `request.accepts(type)`: 检查类型是否可以接受。
- `request.acceptsLanguage(language)`: 检查语言。
- `request.acceptsCharset(charset)`: 检查字符集。
- `request.is(type)`: 检查类型。
- `request.ip`: 读取一个 IP 地址。
- `request.ips`: 读取 IP 地址(使用 `trust-proxy`)。
- `request.path`: 读取 URL 路径。
- `request.host`: 访问主机名, 但不包含端口号。
- `request.fresh`: 检查请求是否是新的。
- `request.stale`: 检查请求是否是旧的。
- `request.xhr`: 检查 XHR/AJAX-y 请求。
- `request.protocol`: 返回 HTTP 协议。
- `request.secure`: 检查协议是否为 `https`。
- `request.subdomains`: 读取子域名数组。
- `request.originalUrl`: 读取原始 URL。

## 响应

- `response.redirect(status, url)`: 重定向请求。
- `response.send(status, data)`: 发送响应。

- `response.json(status, data)`: 发送 JSON 格式的数据, 并强制正确的头信息。
- `response.sendFile(path, options, callback)`: 发送文件。
- `response.render(templateName, locals, callback)`: 渲染模板文件。
- `response.locals`: 将数据传递给模板。

## 处理程序签名

- `function(request, response, next) {}`: 请求处理程序签名。
- `function(error, request, response, next) {}`: 错误处理程序签名。

## Stylus 和 Jade

安装 Jade 和 Stylus:

```
$ npm i -SE stylus jade
```

应用 Jade 模板引擎:

```
app.set('views', path.join(__dirname, 'views'))  
app.set('view engine', 'jade')
```

应用 Stylus CSS 处理器:

```
app.use(require('stylus').middleware(path.join(__dirname, 'public')))
```

## Body

```
var bodyParser = require('body-parser')  
app.use(bodyParser.json())  
app.use(bodyParser.urlencoded({  
  extended: true  
}))
```

## 静态资源

```
app.use(express.static(path.join(__dirname, 'public')))
```

## 连接中间件

```
$ sudo npm install <package_name> --save
```

- `body-parser`(<https://github.com/expressjs/body-parser>): 访问请求负载。
- `compression`(<https://github.com/expressjs/compression>): 使用 Gzip 压缩。

- `connect-timeout`(<https://github.com/expressjs/timeout>): 在指定时间后关闭请求。
- `cookie-parser`(<https://github.com/expressjs/cookie-parser>): 解析和读取 cookie。
- `cookie-session`(<https://github.com/expressjs/cookie-session>): 使用 cookie 存储 session 值。
- `csurf`(<https://github.com/expressjs/csurf>): 为跨站点请求伪造(Cross-Site Request Forgery, CSRF)生成令牌。
- `errorhandler`(<https://github.com/expressjs/errorhandler>): 开发环境使用的错误处理程序。
- `express-session`(<https://github.com/expressjs/session>): 通过内存或其他方法来存储会话。
- `method-override`(<https://github.com/expressjs/method-override>): 覆盖 HTTP 方法。
- `morgan`(<https://github.com/expressjs/morgan>): 输出服务器日志。
- `response-time`(<https://github.com/expressjs/response-time>): 显示响应时间。
- `serve-favicon`(<https://github.com/expressjs/serve-favicon>): 提供一个 favicon。
- `serve-index`(<https://github.com/expressjs/serve-index>): 提供文件目录作为文件服务器。
- `serve-static`(<https://github.com/expressjs/serve-static>): 静态资源服务。
- `vhost`(<https://github.com/expressjs/vhost>): 使用虚拟主机。

## 其他流行的中间件

- `cookie`(<https://github.com/jed/cookies>)和 `keygrip`(<https://github.com/jed/keygrip>): 解析 cookie(类似于 `cookie-parser`)。
- `raw-body`(<https://github.com/stream-utils/raw-body>): 使用原始的 payload/body。
- `connect-multiparty`(<https://github.com/superjoe30/connect-multiparty>): 处理文件上传。
- `qs`(<https://github.com/ljharb/qs>): 解析以对象和数组作为值的查询字符串。
- `st`(<https://github.com/isaacs/st>)和 `connect-static`(<https://github.com/andrewrk/connect-static>): 静态文件服务(类似于 `staticCache`)。
- `express-validator`(<https://github.com/ctavan/express-validator>): 执行验证。
- `less`(<https://github.com/emberfeather/less.js-middleware>): 将 LESS 文件转换为 CSS。
- `passport`(<https://github.com/jaredhanson/passport>): 验证请求。
- `helmet`(<https://github.com/evilpacket/helmet>): 设置安全头。
- `connect-cors`(<https://npmjs.com/package/cors>): 启用跨源资源共享(Cross-Origin Resource Sharing, CORS)。
- `connect-redis`(<http://github.com/visionmedia/connect-redis>): 连接 Redis。

## 资源

- 免费的在线课程 *Express Foundation*(<https://node.university/p/express-foundation>)。
- *Pro Express*(Apress, 2014), 这是一本由我编写的全面介绍 Express 的书, <http://proexpressjs.com>。
- 我在博客上发布的 Express 文章, <https://webapplog.com/tag/express-js>。



## 附录

## D

# MongoDB 和 Mongoose 速查表

开发项目时，在互联网上搜索 React 文档和 API，或者回到本书的某一章，查找一个方法，这些方式的效率并不高。如果想节省时间，避免网络上的其他东西使你分心，建议使用本附录中的 MongoDB 和 Mongoose 速查表作为快速参考。

## 打印 PDF 版本

除了这里介绍的文本版本之外，我还创建了一个免费的设计精美的 PDF 版本。可以通过访问 <http://reactquickly.co/resources> 下载 PDF 版本，如图 D.1 所示。

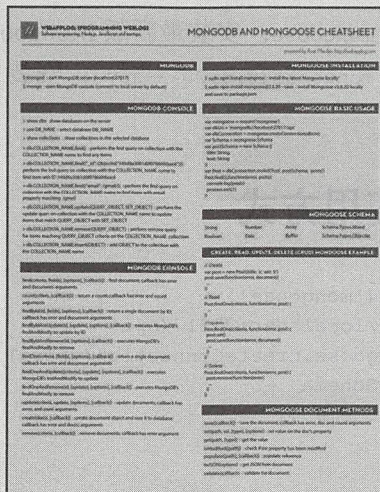


图 D.1 PDF 版本的 MongoDB 和 Mongoose 速查表

## MongoDB

- `$ mongod`: 启动 MongoDB 服务(`localhost:27017`)。
- `$ mongo`(默认连接本地服务器): 打开 MongoDB 控制台。

## MongoDB 控制台

- `> show dbs`: 显示服务器上的数据库列表。
- `> use DB_NAME`: 选择 `DB_NAME` 数据库
- `> show collections`: 显示所选数据库中的集合
- `> db.COLLECTION_NAME.find()`: 对名为 `COLLECTION_NAME` 的集合执行查询, 查找所有项。
- `> db.COLLECTION_NAME.find({ "_id": ObjectId("549d9a3081d0f07866fdaac6") })`: 对名为 `COLLECTION_NAME` 的集合执行, 查找 `id` 为 `549d9a3081d0f07866fdaac6` 的项。
- `> db.COLLECTION_NAME.find({"email":/gmail/})`: 对名为 `COLLECTION_NAME` 的集合执行查询, 查找 `email` 属性匹配/`gmail`/的项。
- `> db.COLLECTION_NAME.update(QUERY_OBJECT, SET_OBJECT)`: 对名为 `COLLECTION_NAME` 的集合执行查询, 查找匹配 `QUERY_OBJECT` 的项并更新为 `SET_OBJECT`。
- `> db.COLLECTION_NAME.remove(QUERY_OBJECT)`: 对 `COLLECTION_NAME` 集合中符合 `QUERY_OBJECT` 条件的项执行删除操作。
- `> db.COLLECTION_NAME.insert(OBJECT)`: 将 `OBJECT` 添加到名为 `COLLECTION_NAME` 的集合中。

## 安装 Mongoose

- `$ sudo npm install mongoose`: 安装最新版 Mongoose。
- `$ sudo npm install mongoose@3.8.20 --save`: 安装 3.8.20 版本的 Mongoose 并添加至 `package.json` 中。

## Mongoose 基本使用方式

```
var mongoose = require('mongoose')
var dbUri = 'mongodb://localhost:27017/api'
var dbConnection = mongoose.createConnection(dbUri)
var Schema = mongoose.Schema
var postSchema = new Schema ({
  title: String,
  text: String
})
```

```
var Post = dbConnection.model('Post', postSchema, 'posts')
Post.find({}, function(error, posts){
  console.log(posts)
  process.exit(1)
})
```

## Mongoose schema

- `Sing`
- `Boolean`
- `Number`
- `Date`
- `Array`
- `Buffer`
- `Schema.Types.Mixed`
- `Schema.Types.ObjectId`

## Mongoose CRUD(创建、读取、更新、删除)示例

```
// Create
var post = new Post({title: 'a', text: 'b'})
post.save(function(error, document){
  ...
})

// Read
Post.findOne(criteria, function(error, post) {
  ...
})

// Update
Post.findOne(criteria, function(error, post) {
  post.set()
  post.save(function(error, document){
    ...
  })
})

// Delete
Post.findOne(criteria, function(error, post) {
  post.remove(function(error) {
    ...
  })
})
```



## Mongoose 模型方法

- `find(criteria, [fields], [options], [callback])`: 查找文档, 其中的 `callback` 含有 `error` 和 `documents` 参数。
- `count(criteria, [callback])`: 返回匹配条件的文档数目, 其中 `callback` 含有 `error` 和 `count` 参数。
- `findById(id, [fields], [options], [callback])`: 通过 `id` 返回单个文档, 其中 `callback` 含有 `error` 和 `document` 参数。
- `findByIdAndUpdate(id, [update], [options], [callback])`: 通过 `id` 查找并执行 MongoDB 的 `findAndModify()` 方法来更新文档。
- `findByIdAndRemove(id, [options], [callback])`: 通过 `id` 查找并执行 MongoDB 的 `findAndModify()` 方法来删除文档。
- `findOne(criteria, [fields], [options], [callback])`: 返回单个文档, 其中 `callback` 含有 `error` 和 `document` 参数。
- `findOneAndUpdate([criteria], [update], [options], [callback])`: 执行 MongoDB 的 `findAndModify()` 方法以更新文档。
- `findOneAndRemove(id, [update], [options], [callback])`: 执行 MongoDB 的 `findAndModify()` 方法以删除一个文档。
- `update(criteria, update, [options], [callback])`: 更新文档, 其中 `callback` 含有 `error` 和 `count` 参数。
- `create(doc(s), [callback])`: 创建一个文档对象, 并将该文档对象保存到数据库中, 其中 `callback` 含有 `error` 和 `doc(s)` 参数。
- `remove(criteria, [callback])`: 删除文档, 其中 `callback` 含有 `error` 参数。

## Mongoose 文档方法

- `save([callback])`: 保存文档, 其中 `callback` 含有 `error`、`doc`、`count` 三个参数。
- `set(path, val, [type], [options])`: 在文档的属性上设置一个值。
- `get(path, [type])`: 获取属性的值。
- `isModified([path])`: 检查属性是否有更改。
- `populate([path], [callback])`: 填充一个引用。
- `toJSON(options)`: 从文档中获取 JSON 格式的数据。
- `validate(callback)`: 验证文档。

# 附录 E

## ES6 简介

本附录简要介绍 ES6，描述新一代最流行编程语言 JavaScript 的 10 个最佳特性：

- 默认参数
- 模板字面量
- 多行字符串
- 解构赋值
- 增强对象字面量
- 箭头函数
- Promise
- 块级作用域：let 和 const
- 类
- 模块化

注意：这是一个非常主观的列表，但并不是说其他 ES6 特性的实用性很低，之所以没有列出其他特性，是因为我想将数量限制在 10 个以内。

### 默认参数

之前，我们使用如下方式定义默认变量：

```
var link = function (height, color, url) {  
  var height = height || 50  
  var color = color || 'red'  
  var url = url || 'http://azat.co'  
  ...  
}
```

当参数值不为 0 时，上述代码是没有问题的；参数值变成 0 以后，程序将会出现问题，会变成后面的硬编码值而不是 0 值本身。这是因为在 JavaScript 中，0 值被认为是假值。当然，谁会用 0 来传值啊？所以我们忽略了这个缺陷，使用了逻辑或( || )操作符。在 ES6 中，不再使用以上方式，我们将默认值直接放在函数签名中：

```
var link = function (height = 50, color = 'red', url = 'http://azat.co') {  
  ...  
}
```

这种语法类似于 Ruby。我喜欢的 CoffeeScript 也有这个功能很多年了。

打印 PDF 版本

除了这里介绍的文本版本之外，我还创建了一个免费的设计精美的 PDF 版本。可以通过访问 <http://reactquickly.co/resources> 下载 PDF 版本，如图 E.1 所示。

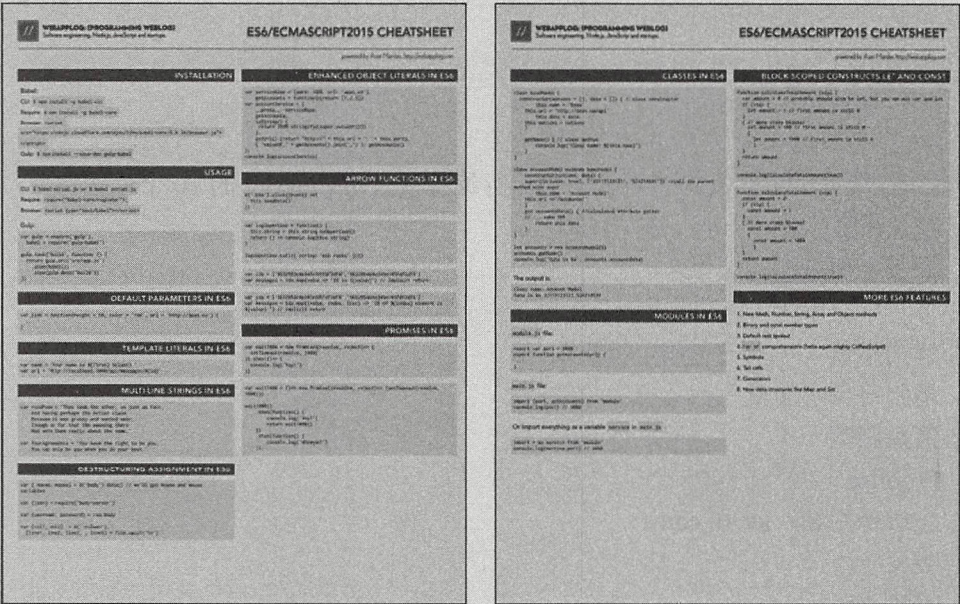


图 E.1 PDF 版本的 ES6/ES2015 速查表

模板字面量

模板字面量或其他语言的字符串插值都可以在字符串中插入变量。在 ES5 中，需要按如下方式把字符串打散拼接：

```
var name = 'Your name is ' + first + ' ' + last + '.'  
var url = 'http://localhost:3000/api/messages/' + id
```

在 ES6 中，可以在反引号包裹的字符串中使用 `${NAME}` 语法：

```
var name = `Your name is ${first} ${last}.`  
var url = `http://localhost:3000/api/messages/${id}`
```



你是否怀疑在 Markdown 中仍然可以使用模板字面量语法？Markdown 对内嵌代码块使用反引号。这是一个问题！解决方法是对有反引号字符串模板的 Markdown 代码使用两个、三个或更多个反引号。

## 多行字符串

另一个美味的语法糖是多行字符串。在 ES5 中，必须使用以下任意方法：

```
var roadPoem = 'Then took the other, as just as fair,\n\t'+ 'And having perhaps the better claim\n\t'+ 'Because it was grassy and wanted wear,\n\t'+ 'Though as for that the passing there\n\t'+ 'Had worn them really about the same,\n\t'\nvar fourAgreements = 'You have the right to be you.\n\tYou can only be you when you do your best.'
```

在 ES6 中，可以使用反引号：

```
var roadPoem = `Then took the other, as just as fair,\n\tAnd having perhaps the better claim\n\tBecause it was grassy and wanted wear,\n\tThough as for that the passing there\n\tHad worn them really about the same,`\nvar fourAgreements = `You have the right to be you.\n\tYou can only be you when you do your best.`
```

## 解构赋值

解构可能是一个比较难理解的概念，因为这里真的有技巧。假如有一个简单的赋值表达式，需要将对象中的属性 `house` 和 `mouse` 赋值给变量 `house` 和 `mouse`：


```
var data = {('body').data(),\n  house = data.house,\n  mouse = data.mouse}
```



data有属性house和mouse

下面是另一个解构赋值示例(在 Node.js 中)：

```
var jsonMiddleware = require('body-parser').json\nvar body = req.body,\n  username = body.username,\n  password = body.password
```



body有username和password属性

在 ES6 中，可以使用如下语句替代上述 ES5 代码：

```
var { house, mouse } = $('body').data()
var { json } = require('body-parser')
var { username, password } = req.body
```

得到 house 和 mouse 变量的值

这种方式在数组中也同样适用，简直疯狂：

```
var [col1, col2] = $('.column'),
    [line1, line2, line3, , line5] = file.split('\n')
```

上述第一行将\$('.column')的第 0 个元素赋值给 col1，将第 1 个元素赋值给 col2。第二个表达式缺少 line4，最终赋值结果如下，其中 fileSplitArray 为 file.split('\n')的结果：

```
var line1 = fileSplitArray[0]
var line2 = fileSplitArray[1]
var line3 = fileSplitArray[2]
var line5 = fileSplitArray[4]
```

习惯解构赋值语法可能需要一些时间，但它毫无疑问是一件甜蜜的糖衣。

## 增强对象字面量

能用对象字面量做的事让人兴奋！从 ES5 中的 JSON 版本，升级到类似于 ES6 中的类。如下是一个典型的包含方法和属性的 ES5 对象字面量：

```
var serviceBase = {port: 3000, url: 'azat.co'},
    getAccounts = function(){return [1,2,3]}

var accountServiceES5 = {
  port: serviceBase.port,
  url: serviceBase.url,
  getAccounts: getAccounts,
  toString: function() {
    return JSON.stringify(this.valueOf())
  },
  getUrl: function() {return "http://" + this.url + ':' + this.port},
  valueOf_1_2_3: getAccounts()
}
```

如果想做得好看一点，可以通过 Object.create() 方法让 serviceBase 成为 accountService ES5 的原型，从而实现继承：

```
var accountServiceES5ObjectCreate = Object.create(serviceBase)
var accountServiceES5ObjectCreate = {
  getAccounts: getAccounts,
  toString: function() {
    return JSON.stringify(this.valueOf())
  },
  getUrl: function() {return "http://" + this.url + ':' + this.port},
  valueOf_1_2_3: getAccounts()
}
```

`accountServiceES5ObjectCreate` 和 `accountServiceES5` 并不相同, 因为对象 `accountServiceES5` 有如下所示的 `__proto__` 属性(见图 E.2)。但对于这个示例, 我们认为它们相似。在 ES6 的对象字面量中, 我们把 `getAccounts:getAccounts` 简化为 `getAccounts`, 不需要冒号。

我们还可以直接用 `__proto__` 属性设置原型, 这样看起来更加合理(不是 `'__proto__'`, `__proto__` 只是一个属性):

```
var serviceBase = {port: 3000, url: 'azat.co'},
    getAccounts = function(){return [1,2,3]}
var accountService = {
  __proto__: serviceBase,
  getAccounts,
```

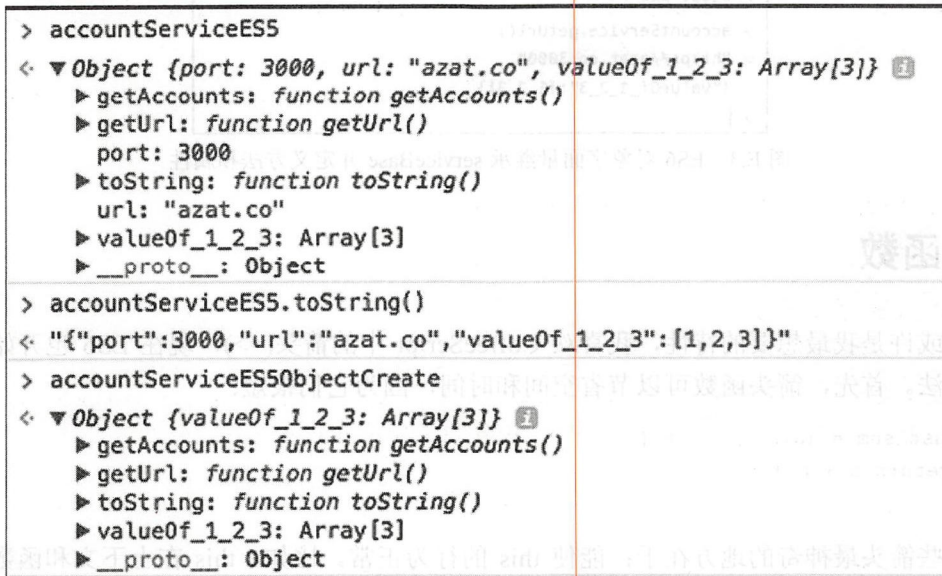


图 E.2 ES5 对象

此外, 我们可以在 `toString()` 中调用 `super` 和动态索引(`valueOf_1_2_3`):

```
toString() {
  return JSON.stringify((super.valueOf()))
},
getUrl() {return "http://" + this.url + ':' + this.port},
```

可以动态创建键、对象属性和属性, 例如使用 `['valueOf_' + getAccounts().join('_')]` 结构创建 `valueOf_1_2_3`:

```
[ 'valueOf_' + getAccounts().join('_') ]: getAccounts()
}
console.log(accountService)
```

图 E.3 中显示了使用 `serviceBase` 作为 `__proto__` 的 ES6 对象 `accountService`, 这是对老旧对象字面量的一次大改进!



```
> accountService
< ▼ Object {valueOf_1_2_3: Array[3]}
  ▶ getAccounts: function getAccounts()
  ▶ getUrl: function getUrl()
  ▶ toString: function toString()
  ▶ valueOf_1_2_3: Array[3]
  ▼ __proto__: Object
    port: 3000
    url: "azat.co"
    ▶ __proto__: Object
> accountService.valueOf_1_2_3
< [1, 2, 3]
> accountService.port
< 3000
> accountService.url
< "azat.co"
> accountService.getUrl()
< "http://azat.co:3000"
  {"valueOf_1_2_3": [1, 2, 3]}
> |
```

图 E.3 ES6 对象字面量继承 serviceBase 并定义方法和属性

## 箭头函数

这或许是我最想要的特性，我喜欢 CoffeeScript 中的箭头(=>)，现在 ES6 也开始支持这种写法。首先，箭头函数可以节省空间和时间，因为它们很短：

```
const sum = (a, b, c) => {
  return a + b + c
}
```

这些箭头最神奇的地方在于：能使 this 的行为正常。比如，this 在上下文和函数中的值应当是相同的，不会发生变化，通常发生变化的原因都是因为创建了闭包。

使用箭头函数意味着不再需要使用 that = this、self = this、\_this = this、和.bind(this)这样的代码。例如，如下丑陋的 ES5 写法：

```
var _this = this
$('.btn').click(function(event) {
  _this.sendData()
})
```

更好的 ES6 代码：

```
$('.btn').click((event) => {
  this.sendData()
})
```

令人遗憾的是，ES6 委员会觉得再加上瘦箭头(->)就太多余了，所以继续使用冗长的 function()。CoffeeScript 中瘦箭头的作用就像 ES5 和 ES6 中的常规函数一样。

这是另一个在 ES5 中使用 call 将上下文传递给 logUpperCase()函数的示例：

```
var logUpperCase = function() {
  var _this = this
```



```

    this.string = this.string.toUpperCase()
    return function () {
      return console.log(_this.string)
    }
  }

  logUpperCase.call({ string: 'es6 rocks' })()

```

在 ES6 中，不再需要 `_this`：

```

var logUpperCase = function() {
  this.string = this.string.toUpperCase()
  return () => console.log(this.string)
}

logUpperCase.call({ string: 'es6 rocks' })()

```

注意，在 ES6 中可以合理地把 `=>` 和 `function` 混用。当箭头函数所在语句只有一行时，它会变成一个表达式，直接返回这条语句的值。但是如果有多行语句，就要显式地使用 `return` 返回。

如下是在 ES5 中利用 `messages` 数组创建数组的代码：

```

var ids = ['5632953c4e345e145fdf2df8', '563295464e345e145fdf2df9']
var messages = ids.map(function (value) {
  return "ID is " + value
})

```

← 显式地返回

ES6 方式如下：

```

var ids = ['5632953c4e345e145fdf2df8', '563295464e345e145fdf2df9']
var messages = ids.map(value => `ID is ${value}`) //

```

← 隐式地返回

注意这段代码使用了字符串模板，这也是我喜欢的 `CoffeeScript` 功能。

在只有一个参数的箭头函数中，圆括号是可有可无的，但是当有多个参数时，就必须使用圆括号。在 ES5 中，以下代码具有显式返回的 `function()`：

```

var ids = ['5632953c4e345e145fdf2df8', '563295464e345e145fdf2df9'];
var messages = ids.map(function (value, index, list) {
  return 'ID of ' + index + ' element is ' + value + ' '
})

```

← 显式地返回

在 ES6 中更具说明性的版本(使用圆括号包裹参数和隐式返回)如下：

```

var ids = ['5632953c4e345e145fdf2df8', '563295464e345e145fdf2df9']
var messages = ids.map((value, index, list) =>
  `ID of ${index} element is ${value}`)

```

← 隐式地返回

## Promise

`Promise` 历来是一个有争议的话题。现在有很多 `promise` 实现，语法略微不同，比如 `Q`、`Bluebird`、`Deferred.js`、`Vow`、`Avow` 和 `jQuery Deferred` 等。其他开发者说，“我们不需要



Promise，可以使用异步、生成器 generator 和回调等。”幸运的是，现在 ES6 中有一个标准的 Promise 实现。

现在我们来查看一个使用 `setTimeout()` 实现异步执行的简单示例：

```
setTimeout(function() {  
  console.log('Yay!')  
}, 1000)
```

使用 ES6 中的 Promise 重写后：

```
var wait1000 = new Promise(function(resolve, reject) {  
  setTimeout(resolve, 1000)  
}).then(function() {  
  console.log('Yay!')  
})
```

也可以使用箭头函数：

```
var wait1000 = new Promise((resolve, reject)=> {  
  setTimeout(resolve, 1000)  
}).then(()=> {  
  console.log('Yay!')  
})
```

到目前为止，我们将代码行数从 3 行增加到 5 行，并没有带来任何明显的好处。但是如果 `setTimeout()` 回调中有更多的嵌套逻辑，好处就来了。如下代码：

```
setTimeout(function() {  
  console.log('Yay!')  
  setTimeout(function() {  
    console.log('Wheeyee!')  
  }, 1000)  
, 1000)
```

使用 ES6 中的 Promise 重写后：

```
var wait1000 = ()=> new Promise((resolve, reject)=>  
  {setTimeout(resolve, 1000)})  
wait1000()  
  .then(function() {  
    console.log('Yay!')  
    return wait1000()  
  })  
  .then(function() {  
    console.log('Wheeyee!')  
  })  
);
```

你还是无法相信 Promise 比常规回调更好？我也不相信。我想一旦你知道回调的方法，就不需要额外复杂的 Promise 了。尽管如此，Promise 是为懂得欣赏它的人准备的，Promise 有不错的错误捕获机制。想查看 Promise 的更多知识，请参阅 James Nelson 的文章“ES6 Promise 简介：用四个方法避免回调地狱”(<http://mng.bz/3OAP>)。





## 块级作用域：let 和 const

你可能已经听说过 let 的一些奇怪说法。它并不是语法糖，它很复杂。let 是一个新的 var，它可以把变量的作用域限制在当前块内。我们用花括号({})来定义块，但是在 ES5 中，块对变量不起任何作用：

```
function calculateTotalAmount (vip) {  
  var amount = 0  
  if (vip) {  
    var amount = 1  
  }  
  { // More crazy blocks!  
    var amount = 100  
    {  
      var amount = 1000  
    }  
  }  
  return amount  
}  
  
console.log(calculateTotalAmount(true))
```

运行结果是 1000。天呐！这是多么大的一个 Bug。在 ES6 中，可以使用 let 严格限制变量作用域在块内。如下变量在的作用域函数范围内：

```
function calculateTotalAmount (vip) {  
  var amount = 0 // Probably should also be let, but you can mix var and let  
  if (vip) {  
    let amount = 1 // First amount is still 0  
  }  
  { // more crazy blocks!  
    let amount = 100 // First amount is still 0  
    {  
      let amount = 1000 // First amount is still 0  
    }  
  }  
  return amount  
}  
  
console.log(calculateTotalAmount(true))
```

结果是 0，因为 if 块中有 let。如果没有 let，结果将是 1。

说到 const，事情就简单多了。它仅仅产生一个只读变量(只读指的是变量不可重新赋值)，并且它的作用域也像 let 一样只在当前块内。const 对对象依然可用，而且对象的属性是可以改变的。

假设有一个常量 url，如 const url = "http://webapplog.com"。在大多数浏览器中，使用 const url = "http://azat.co" 重新分配将会失败——尽管文档规定，const 并不意味着不可变性，但是如果尝试更改它的值，那么它并不会改变。



为了演示，如下代码定义了一些常量，并且由于块级作用域的原因，这些定义都是有效的：

```
function calculateTotalAmount (vip) {  
  const amount = 0  
  if (vip) {  
    const amount = 1  
  }  
  { // More crazy blocks!  
    const amount = 100  
    {  
      const amount = 1000  
    }  
  }  
  return amount  
}  
  
console.log(calculateTotalAmount(true))
```

依我之见，`let` 和 `const` 使得这门语言变得复杂。如果没有它们，我们只有一条路可走，但现在我们需要考虑更多的情况。

## 类

如果喜欢面向对象编程，那么你会喜欢这个特性。它让你在编写类和继承类时，就像在 Facebook 上发布评论那么简单。

在 ES5 中，因为没有 `class` 关键字(但它是毫无作用的保留字)，类的创建和使用是让人十分头疼的事情。此外，很多继承模式(如 `pseude-classical`<sup>1</sup>和 `classical`<sup>2</sup>)，泛函让人越来越感到凌乱困惑，为 JavaScript 的信仰战争火上浇油。

我不会向你展示在 ES5 中怎么去编写一个类(是的，从对象可以衍生出其他的类和对象)，因为有太多方法能够实现类。我们直接看 ES6 示例，ES6 的类用原型而不是工厂方法来实现。现在有一个 `baseModel` 类，其中定义了一个构造函数和 `getName()` 方法：

```
class baseModel {  
  constructor(options = {}, data = []) { ← 类的构造函数  
    this.name = 'Base'  
    this.url = 'http://azat.co/api'  
    this.data = data  
    this.options = options  
  }  
  getName() {  
    console.log(`Class name: ${this.name}`)  
  }  
}
```

类方法 →

1 Ilya Kantor, “Class Patterns,” <http://javascript.info/class-patterns>

2 Douglas Crockford, “Classical Inheritance in JavaScript,” [www.crockford.com/javascript/inheritance.html](http://www.crockford.com/javascript/inheritance.html)



注意以上代码为 `options` 和 `data` 使用了默认参数，而且方法名也不再需要加上 `function` 或冒号(`:`)。另一个很大的区别是：不能像方法那样分配属性(`this.NAME`)。也就是说，不能与方法在同一缩进级别声明 `name`。如果需要设置属性的值，请在构造函数中分配。

`AccountModel` 使用语法 `class NAME extends PARENT_NAME` 继承 `baseModel`。可以很轻松地使用带有参数的 `super()` 调用父类的构造函数：

```
class AccountModel extends baseModel {
  constructor(options, data) {
    super({private: true}, ['32113123123', '524214691'])
    this.name = 'Account Model'
    this.url += '/accounts/'
  }
}
```

使用 `super()` 调用父类构造函数

如果想要高级一点的话，可以像下面这样定义一个 `getter`，这样 `accountsData` 就会变成一个属性：

```
class AccountModel extends baseModel {
  constructor(options, data) {
    super({private: true}, ['32113123123', '524214691'])
    this.name = 'Account Model'
    this.url += '/accounts/'
  }
  get accountsData() {
    // ... make XHR
    return this.data
  }
}
```

返回计算后的数据

现在如何使用这个技巧呢？很简单：

```
let accounts = new AccountModel(5)
accounts.getName()
console.log('Data is %s', accounts.accountsData)
```

输出结果如下：

```
Class name: Account Model
Data is %s 32113123123,524214691
```

## 模块化

如你所知，JavaScript 在 ES6 之前并不支持原生模块化。人们想出了 AMD、RequireJS、CommonJS 和其他解决方法。现在终于有了 `import` 和 `export` 运算符来支持模块化。

在 ES5 中，可以使用带有 IIFE(Immediately Invoked Function Expression，立即执行函数表达式)的 `script` 标签，或是其他的诸如 AMD 之类的库，但是在 ES6 中可以用 `export` 来暴露类。我喜欢 Node.js，所以我使用和 Node.js 语法一样的 CommonJS。

使用 Browserify 编译并打包，使得在浏览器上使用 CommonJS 非常简单。现在，ES5 文件 `module.js` 中有一个 `port` 变量和 `getAccounts` 方法：





```
module.exports = {
  port: 3000,
  getAccounts: function() {
    ...
  }
}
```

在 ES5 文件 `main.js` 中, 使用 `require('module')` 导入依赖:

```
var service = require('module.js')
console.log(service.port) // 3000
```

在 ES6 中, 可以使用 `export` 和 `import`。比如, 如下是 ES6 中的 `module.js`:

```
export var port = 3000
export function getAccounts(url) {
  ...
}
```

在 ES6 文件 `main.js` 中, 使用语法 `import {name} from 'my-module'` 导入依赖:

```
import {port, getAccounts} from 'module'
console.log(port) // 3000
```

或者直接在 `main.js` 中引入所有内容为 `service` 变量:

```
import * as service from 'module'
console.log(service.port) // 3000
```

就个人而言, 我发现 ES6 模块令人困惑。是的, 它们更传神一些, 但 Node.js 模块并不会很快改变。浏览器和服务器最好统一风格, 所以现在我坚持使用 CommonJS/Node.js 的风格。此外, 在撰写本书时, 浏览器并不支持 ES6 模块, 因此需要使用 `jspm`(<http://jspm.io>) 这样的工具来支持 ES6 模块。

想要了解 ES6 更多中的模块化和例子, 请参考文章 [http://exploringjs.com/es6/ch\\_modules.html](http://exploringjs.com/es6/ch_modules.html)。总之, 无论如何, 编写模块化的 JavaScript 吧!

## 通过 Babel 使用 ES6

要想现在使用 ES6, 需要使用 Babel 作为构建过程的一部分。有关 Babel 的更多信息, 请参见第 3 章。

## 其他 ES6 特性

ES6 还有许多其他可能用不上(至少不会马上使用)但值得注意的特性, 以下没有特定的顺序:

- Math/Number/String/Array/Object 中的新方法
- 二进制和八进制数据类型



- 自动展开多余参数
- for of 循环(又见面了, CoffeeScript)
- Symbols
- 尾部调用优化
- 生成器
- 新的数据结构(如 Map 和 Set)

ECMAScript 可以提高工作效率, 减少错误。它还将继续演变发展, 我们对它的学习也不会停止。充分利用下面这些资源:

- ES6 速查表, <http://reactquickly.co/resources>。
- Nicholas C. Zakas 编写的 *Understanding ECMAScript6*(Learnpub, 2017), <https://leanpub.com/understandings6>。
- Axel Rauschmayer 编写的 *Exploring ES6*(Learnpub, 2017), <http://exploringjs.com/es6.html>。
- ES6 课程, <https://node.university/p/es6>。
- ES7 和 ES8 课程, <https://node.university/p/es7-es8>。

# 快速上手React编程

React Quickly

## 技术简介

成功的用户界面需要在视觉上有趣、快速并且流畅。React.js通过改进UI组件之间的数据流来驱动重视图Web应用。React站点可以高效、平稳地更新视觉元素，最大限度地减少页面重载。React对开发者友好，具有强大的生态系统来支持整个应用栈的开发过程，并且因为全部使用JavaScript，可以快速上手。

## 内容简介

本书面向希望快速上手React.js进行Web开发的读者，借助精心挑选和详细解释的实例，帮助读者使用现有的JavaScript和Web开发技能学习React开发。在学习Web组件、表单和数据的过程中，还将探索许多不同的项目。

## 主要特点

- 掌握React基础
- 使用数据和路由构建完整的Web应用
- 测试组件
- 优化React应用

## 读者对象

本书的目的是让开发人员舒适地使用JavaScript构建Web应用。

## 作者简介

**Azat Mardan**是供职于第一资本(Captital One)的一名技术研究员，拥有丰富的JavaScript和Node使用和培训经验，还曾撰写多本关于JavaScript、Node、React和Express的书籍。

“学习React.js的最好方法。”

——John Sonmez,  
Soft Skills作者

“对于任何想要获得React入门指导，并了解React周边生态系统(包括工具、概念和库)的读者来说，本书提供了一种一站式服务。”

——Peter Cooper,  
JavaScript Weekly编辑

“非常适合React开发新手和经验丰富的老兵。”

——Matthew Heck,  
TechChange

“绝对有吸引力，理论结合实践。”

——Dane Balia,  
Hetzner

“React最佳快速入门。”

——Art Bergquist,  
Cognetic Technologies



源代码下载

清华社官方微信号



扫我有惊喜

ISBN 978-7-302-50247-0



9 787302 502470 >

定价：88.00元